UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Miloš Pisarević

# Primerjava brezizgubnih algoritmov za stiskanje podatkov s sistemom ALGator

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: doc. dr. Tomaž Dobravec

Ljubljana, 2016

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Na področju računalništva je stiskanje podatkov zelo pomemben in pogosto uporabljen postopek, s katerim zmanjšamo velikost predstavitve vhodnih podatkov.

V diplomskem delu preučite in predstavite različne tehnike stiskanja podatkov. Osredotočite se na statistične algoritme, na algoritme za stiskanje s pomočjo slovarja ter na algoritem Burrows-Wheeler. Najpomembnejše predstavnike posamezne skupine natančno opišite ter implementirajte v programskem jeziku Java. Algoritme preizkusite na znanih množicah testnih datotek (Calgary, Canterbury ter Silesia). Za testiranje pravilnosti, kakovosti in hitrosti implementiranih algoritmov uporabite sistem ALGator.

# IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani Miloš Pisarević, z vpisno številko 63110110, sem avtor diplomskega dela z naslovom:

*Primerjava brezizgubnih algoritmov za stiskanje podatkov s sistemom ALGator*

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Tomaža Dobravca,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu prek univerzitetnega spletnega arhiva.

V Ljubljani, dne 14. marca 2016 Podpis avtorja:

*Ob tej priložnosti bi se rad zahvalil mentorju doc. dr. Tomažu Dobravcu za strokovno pomoč, usmerjanje, nasvete in potrpežljivost pri izdelavi diplomskega dela. Posebno zahvalo bi rad naklonil staršem in sestri za vso podporo v času študija.*

# Kazalo

# Seznam uporabljenih kratic

| Kratica | Angleško | Slovensko |
|---|---|---|
| **ASCII** | American Standard Code for Information Interchange | Standardni nabor kod za izmenjavo informacij |
| **BWT** | Burrows-Wheeler transform | Transformacija Burrows-Wheeler |
| **CCITT** | Consultative Committee for International Telephony and Telegraphy | Mednarodna posvetovalna komisija za telefonijo in telegrafijo |
| **DICOM** | Digital Imaging and Communications in Medicine | Standard za izmenjavo in uporabo medicinskih slik |
| **EC** | Entropy coding | Entropijsko kodiranje |
| **EOF** | End of File | Znak za konec datoteke |
| **GIF** | Graphics Interchange Format | Bitni slikovni format |
| **GNU** | GNU's Not Unix | Izvedenka operacijskega sistema Unix |
| **HTML** | HyperText Markup Language | Spletni označevalni jezik |
| **HTTP** | Hypertext Transfer Protocol | Protokol za prenos hiperteksta |
| **IID** | Independent and identically distributed | Neodvisno in identično porazdeljeno |
| **JPEG** | Joint Photographic Experts Group | Bitni slikovni format |
| **LZ77** | Lempel-Ziv 77 | Algoritem Lempel-Ziv 77 |
| **LZ78** | Lempel-Ziv 78 | Algoritem Lempel-Ziv 78 |
| **LZSS** | Lempel-Ziv-Storer–Szymanski | Algoritem Lempel-Ziv-Storer–Szymanski |
| **LZW** | Lempel-Ziv-Welch | Algoritem Lempel-Ziv-Welch |
| **MTF** | Move-to-front | Transformacija premakni-na-začetek |
| **PDF** | Portable Document Format | Format za zapisovanje dokumentov |
| **PNG** | Portable Network Graphics | Bitni slikovni format |
| **RLE** | Run-length encoding | Kodiranje zaporedij enakih znakov |
| **SAO** | Smithsonian Astrophysical Observatory | Raziskovalni inštitut v Cambridgeu |
| **SDF** | Structure-data file | Format za shranjevanje podatkov o strukturi kemikalij |
| **SPARC** | Scalable Processor Architecture | Mikroprocesorska arhitektura |
| **TIFF** | Tagged Image File Format | Bitni slikovni format |
| **VAX** | Virtual Address Extension | Nabor inštrukcij računalniške arhitekture |
| **VLC** | Variable-length code | Koda spremenljive dolžine |
| **XML** | Extensible Markup Language | Razširljiv označevalni jezik |

# Povzetek

**Naslov:** Primerjava brezizgubnih algoritmov za stiskanje podatkov s sistemom ALGator

Namen diplomskega dela je bil predstaviti temeljne algoritme za brezizgubno stiskanje podatkov in jih med seboj testirati. V teoretičnem delu najprej na splošno govorimo o značilnostih kodiranja podatkov. Tukaj razjasnimo razlike med več vrstami kod in se podrobneje osredotočimo na kode VLC. Omenjena je tudi informacijska teorija, pri čemer razložimo, zakaj je pomemben element na področju kodiranja. Nato se poglobimo v razlago dejanskih algoritmov, in sicer začnemo s statističnimi algoritmi, kjer opišemo delovanje Shannon-Fanojevega kodiranja, Huffmanovega kodiranja in aritmetičnega kodiranja. Potem preidemo na algoritme, ki dosegajo stiskanje s pomočjo slovarja, in opišemo prvotna algoritma LZ77 in LZ78 ter njuni varianti LZSS in LZW. Za konec opišemo še algoritem Burrows-Wheeler, ki združuje transformaciji Burrows-Wheeler in premakni-na-začetek z entropijskim kodiranjem. V praktičnem delu začnemo z implementacijo algoritmov Huffmanovega kodiranja, aritmetičnega kodiranja, LZSS, LZW in Bzip2. Algoritme implementiramo z odprtokodnimi implementacijami v programskem jeziku Java in za testiranje uporabimo sistem ALGator. Teste izvajamo na več znanih testnih zbirkah, ki so nastale z namenom testiranja brezizgubnega stiskanja. Te zbirke so Calgary, Canterbury, Silesia in Maximum. Po končanem testiranju rezultate primerjamo med seboj in ugotovimo učinkovitost implementiranih algoritmov.

**Ključne besede:** algoritem, stiskanje, razširjanje, podatki, ALGator, Java, Huffman, Lempel-Ziv, Burrows-Wheeler

# Abstract

**Title:** Comparing lossless data compression algorithms using the ALGator system

The purpose of the thesis is to present the fundamental lossless data compression algorithms, test and compare them. The theoretical part outlines some of the general characteristics of data coding and explores in more detail the differences between different types of codes, particularly VLC codes. Moreover, it touches on the information theory and explains why it plays an important role in coding. It also thoroughly explains actual algorithms, starting with statistical algorithms to describe the functioning of Shannon-Fano coding, Huffman coding and arithmetic coding. Next are algorithms that achieve compression using a dictionary, i.e. algorithms LZ77 and LZ78 and their respective variants LZSS and LZW. The last algorithm presented is the Burrows-Wheeler algorithm that involves the Burrows-Wheeler transformation and move-to-front transformation with entropy coding. As for the practical part, it deals with the implementation and testing of algorithms for Huffman coding, arithmetic coding, LZSS, LZW and Bzip2. The algorithms are implemented with open source implementations in the Java programming language. The ALGator system is used for testing them. The tests are performed on several widely known test collections that were developed for testing lossless compression. Those collections are Calgary, Canterbury, Silesia and Maximum. The comparison of test results determines the effectiveness of the implemented algorithms.



**Keywords:** algorithm, compression, decompression, data, ALGator, Java, Huffman, Lempel-Ziv, Burrows-Wheeler

# 1 Uvod

Stiskanje podatkov je proces pretvarjanja vhodnega toka podatkov v izhodni tok podatkov manjše velikosti s pogojem, da lahko pretvorimo stisnjeni tok nazaj v originalno obliko. Podatkovni tok je lahko v obliki datoteke, medpomnilnika ali posameznih bitov v komunikacijskem toku.

Področje stiskanja podatkov drugače imenujemo tudi t. i. izvorno kodiranje (angl. source coding). Za vsak vhodni znak si predstavljamo, da so oddani od nekega informacijskega izvora, ki jih je treba zakodirati, preden jih pošljemo njihovim ciljem. Znaki so lahko v obliki znakov ASCII, bitov, zvočnih vzorcev, vrednosti pikslov ipd. Izvor ni nujno shranjen v nekem spominu, lahko je brezspominski. V primeru, da je izvor zapisan v spominu, je vsak znak odvisen od predhodnih znakov, mogoče tudi od naslednjih znakov, torej gre za korelacijo. Če je pa izvor brez spomina, potem so znaki neodvisni od predhodnih znakov.

Obstajajo številni znani algoritmi za stiskanje podatkov. Temeljijo na različnih idejah, kjer je vsak primernejši za različne vrste podatkov, zato ustvarjajo različne rezultate, vendar imajo vsi isti cilj, in sicer zmanjšati velikost podatkov z odstranjevanjem redundance iz izvora oziroma izvorne datoteke. Dokler gre za nenaključne podatke, ima podatek neko obliko strukture, ki jo lahko izkoristimo in predstavimo dani podatek v manjši obliki. Odstranjevanje redundance oziroma odvečnih podatkov je ključni pojem, kadar govorimo o stiskanju podatkov.

Z leti je stiskanje podatkov postalo velik vidik računalništva, in sicer tako velik, da je J. Gerard Wolff predlagal novo teorijo [19] računalništva, ki temelji na konceptu stiskanja podatkov z imenom SP Theory of intelligence. Teorija pravi, da lahko stiskanje podatkov razlagamo kot proces odstranjevanja nepotrebnih kompleksnosti v informacijah in s tem maksimiziramo enostavnost, medtem ko ohranimo, kolikor se le da, njeno neodvečno opisno moč.

V računalništvu rečemo, da so podatki sestavljeni iz končne množice zaporedja elementov, ki jo imenujemo abeceda. Elemente abecede, ki predstavljajo vse možne elemente abecede, imenujemo simboli ali znaki. Glavna lastnost abecede je njena velikost, ki nam pove število znakov, iz katerih je abeceda sestavljena. Velikost abecede je odvisna od vrste podatkov. Npr.

pri zaporedju boolean je abeceda sestavljena iz dveh znakov, `true` in `false`, kar je predstavljeno z enim bitom. Za tipično angleško besedilo je abeceda sestavljena iz 128 znakov, ki jih predstavimo s 7-bitnimi kodami ASCII.

| Črka | Frekvenca (%) | Črka | Frekvenca (%) |
|---|---|---|---|
| e | 10,72 | p | 3,37 |
| a | 10,48 | m | 3,30 |
| o | 9,09 | z | 2,10 |
| i | 9,05 | b | 1,94 |
| n | 6,33 | u | 1,88 |
| l | 5,27 | g | 1,63 |
| s | 5,06 | č | 1,48 |
| r | 5,01 | h | 1,04 |
| j | 4,68 | š | 0,99 |
| t | 4,33 | c | 0,66 |
| k | 3,70 | ž | 0,64 |
| v | 3,62 | f | 0,11 |
| d | 3,39 | | |

Tabela 1.1: Frekvenca pojavitve črk v slovenskem besedilu.

V naravnem jeziku se posamezni glasovi oziroma črke ne pojavljajo enako pogosto, lahko se celo pojavljajo le v določenih kombinacijah. Iz Tabela 1.1 vidimo, da sta v slovenskem jeziku najbolj uporabljeni črki `e` in `a`, medtem ko se črka `f` zelo redko pojavlja [11]. Tej razliki frekvenc črk rečemo abecedna redundanca, ki predlaga, da namesto enako dolgih kod ASCII pogostejšim črkam raje dodelimo krajše kode, redkim črkam pa daljše dolžine kod. S tem bi črka `e` dobila najkrajšo kodo, črka `f` pa najdaljšo. 2 podrobneje opisuje načela kodiranja. Še en primer redundance se kaže kot kontekstualna redundanca, ki jo prikažemo z dejstvom, da v tekstovnih besedilih za znakom vejice skoraj zmeraj sledi le znak presledek.

Nekateri algoritmi za stiskanje podatkov so izgubni. Taki algoritmi dosežejo boljšo učinkovitost pri stiskanju, vendar izgubijo nek del informacij. Ko razširimo stisnjeni tok, rezultat ni enak kot originalni podatkovni tok. Izgubne algoritme je smiselno uporabiti takrat, kadar stiskamo slike, video posnetke ali zvočne posnetke. Če je izguba podatkov majhna, potem najverjetneje ne bomo niti opazili sprememb. Vendar nam za stiskanje tekstovnih datotek in programskih datotek ne bi koristil izgubni algoritem, saj lahko tudi ob majhni spremembi programske datoteke kompletno uniči delovanje celotnega programa. Take datoteke zato stiskamo z brezizgubnimi algoritmi, katerih rezultat razširjanja je zmeraj enak prvotnemu podatkovnemu toku.

# 2 Značilnosti kodiranja

## 2.1 Koda

Koda je znak, ki nadomešča nek drugi znak. Matematično gledano je koda uporabljena za preslikavo znakov, tako da vsakemu posameznemu znaku ali nizu znakov dodelimo kodno besedo oziroma kodo. V računalništvu so kode zapisane v obliki niza bitov. Proces preslikave znakov v kode imenujemo kodiranje, obratni proces pa dekodiranje. Kode so lahko:

- statične ali dinamične,
- fiksne dolžine (angl. fixed-length code) ali spremenljive dolžine (angl. variable-length code, VLC).

Statična koda se zgradi enkrat in se nikoli ne spremeni. Primer takih kod so kode ASCII in Unicode. Dinamična koda se spreminja s časom, ko se vse več in več podatkov procesira. Adaptivno Huffmanovo kodiranje je primer algoritma, ki uporablja dinamične kode. Fiksne kode so znane tudi kot bločne kode. Take kode imajo zmeraj isto dolžino, tudi v primeru, če so dinamične in se spreminjajo. Spremenljive kode oziroma kode VLC imajo različno dolžino, kjer krajše kode predstavljajo pogosto pojavljajoče znake in daljše kode manj pojavljajoče znake.

## 2.2 Kode VLC

Kode VLC so zasnovane glede na distribucijo verjetnosti pojavitve znakov celotne abecede. To storimo v fazi modeliranja, nato pa sledi faza kodiranja podatkov. Vsak tip podatkov ima različno distribucijo verjetnosti, zaradi česar jim lahko ustrezajo drugačne kode. Primer algoritma, ki uporablja kode VLC, je Huffmanovo kodiranje. Pri uporabi kod VLC moramo priložiti njeno kodno tabelo v izhodnem toku kompresije, saj dekodirnik ne pozna verjetnostne distribucije pojavitve podatkov in zato ne more sam zgraditi originalnega sporočila brez kodne tabele. V Tabela 2.1 je prikazana razlika med fiksnimi kodami in kodami VLC, ki smo jih zgradili z verjetnostmi iz Tabela 1.1.

| Znak | a | b | c | č | d | e | f | g |
|---|---|---|---|---|---|---|---|---|
| Fiksna dolžina | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| Spremenljiva dolžina | 11 | 1011 | 101001 | 10101 | 1011 | 0 | 101000 | 1000 |

Tabela 2.1: Kodna tabela fiksnih in spremenljivih dolžin prvih 8 črk.

Če imamo vnaprej podano kodno tabelo, je kodiranje nekega niza v kode VLC predvsem enostavno. Ne potrebujemo nobenih posebnih metod ali postopkov. Program prebere prvotni znak $a_i$ iz vhodnega toka enega za drugim in vsak znak $a_i$ zamenja s pripadajočo kodo $c_i$. Posamezne kode združimo v en sam dolgi niz kod in zapišemo v izhodni tok.

Pri dekodiranju se stvar zaplete, ker mora pri branju posameznih bitov iz vhodnega toka dekodirnik vedeti, koliko je dolga koda ali kdaj se dana koda konča. Zato je pomembno, da pazljivo izberemo primerno množico kod VLC. Prav tako mora imeti dekodirnik na razpolago celotno kodno tabelo ali pa mora znati postopek za pretvorbo kod v prvotne znake.

### 2.2.1 Enolične kode

Zaradi posledice uporabe kod VLC, če ne izberemo primerne kode, se lahko zgodi, da ne moremo enolično razlikovati med nekaterimi znaki.

**Primer 2.1**: Imamo dano množico kod VLC (`0`, `10`, `010`, `101`) za množico znakov (`a`, `b`, `c`, `d`). Iz vhodnega toka preberemo zakodirano sporočilo `0100101010`.

Ko dekodirnik prebere začetni bit `0`, iz bitne tabele razbere, da gre za znak `a`, a hkrati vidi, da gre lahko za del kode `010` znaka `c`. Tukaj nastane problem, ker koda `010` lahko predstavlja znak `c` ali niz znakov `ab`. Zakodiranega sporočila se ne da enolično dekodirati, saj se sporočilo lahko dekodira v `abcda` ali `cadc`. Taka množica kod zato ne ustreza za kodiranje. Ta problem bi sicer lahko rešili z dodatnim ločilnim znakov pri kodiranju sporočila. Npr., če bi za ločilo uporabili znak `/`, bi niz `abcda` zakodirali kot `0/10/010/101/0`. S pomočjo ločila bi dekodirnik točno vedel, kako enolično dekodirati sporočilo, ampak tak način kodiranja ne bi bil praktičen, ker bi morali vsaki kodi dodati dodaten znak, s čimer bi precej izgubili učinkovitost stiskanja. Če je neka koda predpona drugi kodi, kot v Primer 2.1, to še ne pomeni, da se kode ne da enolično dekodirati. Tak primer imamo podan kasneje v Primer 2.2, kjer imamo kodo, ki je predpona drugi kodi in se jo da enolično dekodirati. Ali je množica kod enolična, lahko izvemo s pomočjo algoritma Sardinas-Patterson [15].

### 2.2.2 Takojšnje kode

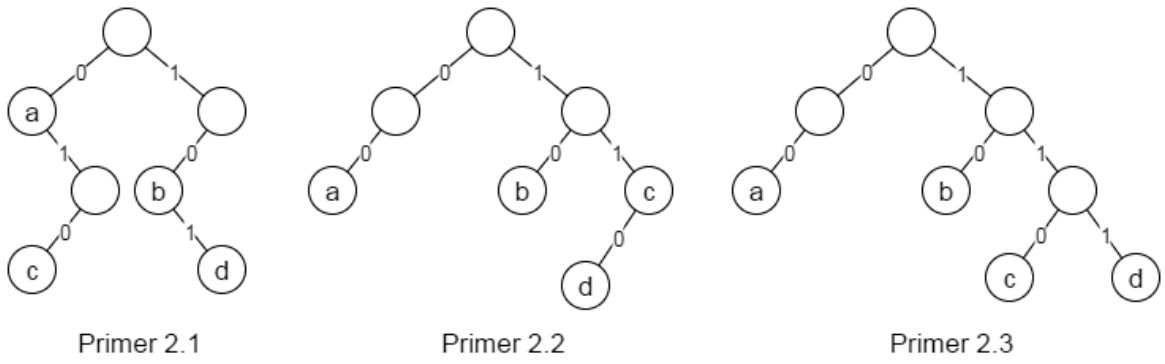
Naslednja pomembna lastnost kod VLC je, da so t. i. takojšnje kode. Ta lastnost ne samo da zmanjša kompleksnost algoritma, temveč tudi prispeva k učinkovitosti stiskanja.

**Primer 2.2**: Imamo množico kod VLC (`00`, `10`, `11`, `110`) za množico znakov (`a`, `b`, `c`, `d`). Sporočilo `dcabc` zakodiramo kot `11011001011`.

Tokrat se da niz enolično dekodirati, vendar dekodiranje še zmeraj ni takojšnje. Pri prvem bitu `1` dekodirnik ugotovi, da gre lahko za znak `b`, `c` ali `d`, zato nadaljuje z branjem. Drugi prebrani bit je zopet `1` in zdaj vidi iz bitne tabele, da gre za znak `c`, a hkrati je lahko tudi nedokončana koda znaka `d`, zato mora vseeno nadaljevati z branjem naslednjih bitov, da ugotovi, ali nato sledi bit `0`. Šele takrat lahko namreč določi, kateri znak predstavlja koda.

**Primer 2.3**: Imamo malo drugačno množico kod VLC (`00`, `10`, `110`, `111`) za množico znakov (`a`, `b`, `c`, `d`). Isto sporočilo `dcabc` zakodiramo kot `1111100010110`.

Za razliko od Primer 2.2 so tukaj kode takojšnje, ker pri branju bitov, ko pridemo do katerega koli možnega znaka, ta znak predstavlja edino možnost. To dejstvo lahko prikažemo z drevesom na Slika 2.1, kjer so vsi znaki Primer 2.3 postavljeni v listih drevesa, drevo iz Primer 2.2 pa vsebuje znak `c` v vozlišču drevesa.

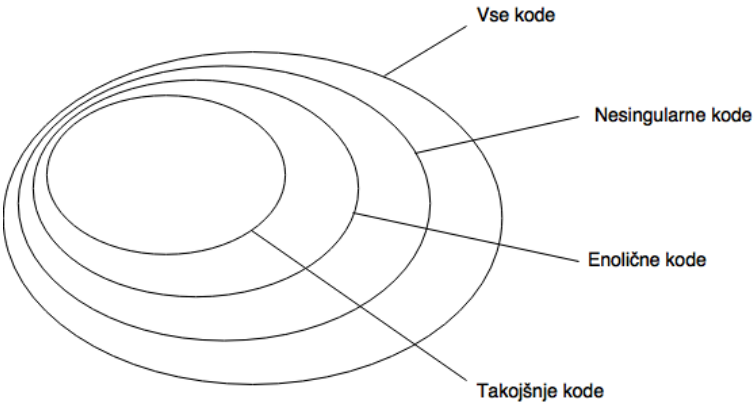



Slika 2.1: Prikaz dreves za Primer 2.1, Primer 2.2 in Primer 2.3.

### 2.2.3 Predponska lastnost

Predponska lastnost (angl. prefix property) pravi, da kadar dodelimo kodo $c_i$ nekemu znaku $a_i$, nobena druga koda $c_j$ ne sme biti začetna predpona kodi $c_i$. Če uporabimo dvojiško drevo, morajo biti kode s predponsko lastnostjo v listih drevesa. Če ima neka koda

predponsko lastnost, to pomeni, da je enolična in takojšnja. Na Slika 2.2 vidimo relacijo med različnimi vrstami kod.



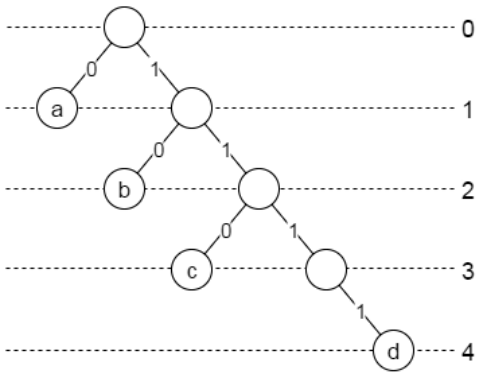

Slika 2.2: Relacija med kodami.

### 2.2.4 Kraftova neenačba

Sama predponska lastnost zagotavlja pravilnost in učinkovitost dekodiranja kod VLC. Da dosežemo dobro kompresijo podatkov, je pomembno tudi, da so vse kode čim krajše.

**Primer 2.4**: Imamo množico kod s predponsko lastnostjo (`0`, `10`, `110`, `1111`) za znake (`a`, `b`, `c`, `d`). Določimo dolžine kod.

Dolžine kod so 1, 2, 3, 4. Dolžino kode lahko razberemo tudi iz drevesa na Slika 2.3, kjer globina vozlišča v drevesu predstavlja dolžino kode. Opazimo, da lahko zadnjo kodo (`1111`) skrajšamo in jo nadomestimo s kodo `111`. Zdaj so kode dolžine 1, 2, 3, 3, za katere vemo, da imajo predponsko lastnost, kajti nobena koda ni predpona drugi kodi. Vendar ne vemo, ali so te dolžine najkrajše možne ob zahtevi, da ohranimo predponsko lastnost.




Slika 2.3: Globina drevesa za Primer 2.4.

**Primer 2.5**: Imamo množico kod s predponsko lastnostjo (0, 10, 110, 111) za znake (a, b, c, d). Dolžine kod so 1, 2, 3, 3. Ali lahko dolžine kod skrajšamo na 1, 2, 2, 3 in hkrati obdržimo predponsko lastnost?

Za odgovor si lahko pomagamo s Kraftovo neenačbo (angl. Kraft's inequality), s katero ugotovimo minimalne dolžine kod s predponsko lastnostjo. Nek niz predponskih kod C = ($c_1$, $c_2$, ..., $c_n$) z n kodami dolžine $l_1$, $l_2$, ..., $l_n$ obstaja samo, če zadošča neenačbi:

$$K(C) = \sum_{j=1}^{n} 2^{-lj} \leq 1 \tag{2.1}$$

Iz rezultata Kraftove neenačbe lahko ugotovimo:

- če Kraftova neenačba drži s strogo neenakostjo (< 1), potem koda vsebuje redundanco v obliki predolgih kod;
- če Kraftova neenačba drži z enakostjo (= 1), potem je koda optimalne dolžine;
- če Kraftova neenačba ne drži (> 1), potem koda ni enolična.

Če uporabimo Kraftovo neenačbo za Primer 2.5, dobimo:

$$K(C) = \sum_{j=1}^{n} 2^{-lj} = \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^2} + \frac{1}{2^3} = 1{,}125 > 1 \tag{2.2}$$

Vidimo, da je rezultat večji od 1 in s tem ne zadošča neenačbi, torej ni mogoče, da zmanjšamo dolžine kod.

Če neka neznana množica dolžin $l_i$ zadošča Kraftovi neenačbi, potem zagotovo vemo, da obstaja neka množica takojšnjih in enoličnih kod VLC z danimi dolžinami. Še ena lastnost neenačbe je, da nam lahko pove, ali kakšna dana koda ni predponska koda, vendar nam ne more povedati, ali je dana koda predponska koda.

## 2.3 Informacijska teorija

Informacijska teorija (angl. Information theory) je študija informacij na osnovi verjetnostne teorije. Teorijo je predlagal Claude E. Shannon [16] leta 1948. Pokazal je matematični način merjenja količine informacij, vsebovan v nekem podatku. Informacija je nekaj, kar doprinese znanju osebe in je zapisana v obliki neke vrste podatkov. Bolj kot sporočilo izraža nekaj, kar

nam je neznano, bolj je sporočilo informativno. Ali drugače, nepričakovanost oziroma nepredvidljivost je sorazmerno s količino informacij v sporočilu. Količino informacij lahko tako merimo tudi kvantitativno v skladu z nepričakovanostjo oziroma s presenečanjem dogodkov [13]. V praktičnem primeru kodiranja, dogodek je enako, kot branje novega znaka iz vhodnega toka.

Npr., da imamo n dogodkov za abecedo S = ($s_1$, $s_2$, ..., $s_n$). Za vsak dogodek $s_j$ imamo verjetnost pojavitve $p_i$. Seštevek verjetnosti dogodkov P = ($p_1$, $p_2$, ..., $p_n$) je enako 1. Domnevamo tudi, da je izvor informacij brezspominski, torej je vsak dogodek oziroma znak neodvisen od ostalih. Količino presenečanja ali informacij dogodka $s_i$ definiramo z matematično enačbo:

$$I(s_i) = log_b \frac{1}{p_i} \tag{2.3}$$

ali z negativnim logaritmom:

$$I(s_i) = -log_b p_i \tag{2.4}$$

Odvisno od osnove logaritma imenujemo enoto informacije:

- bit (b = 2),
- nat (b = e),
- hartley (b = 10).

**Primer 2.6**: Imamo znak $s_i$ = d z verjetnostjo pojavitve $p_i = \frac{1}{4}$. Izračunajmo količino informacij znaka $s_i$.

$$I(s_i) = -log_{2p_i} = -log_2 \frac{1}{4} = -(-log_2 2^2) = 2log_2 2 = 2\ bita \tag{2.5}$$

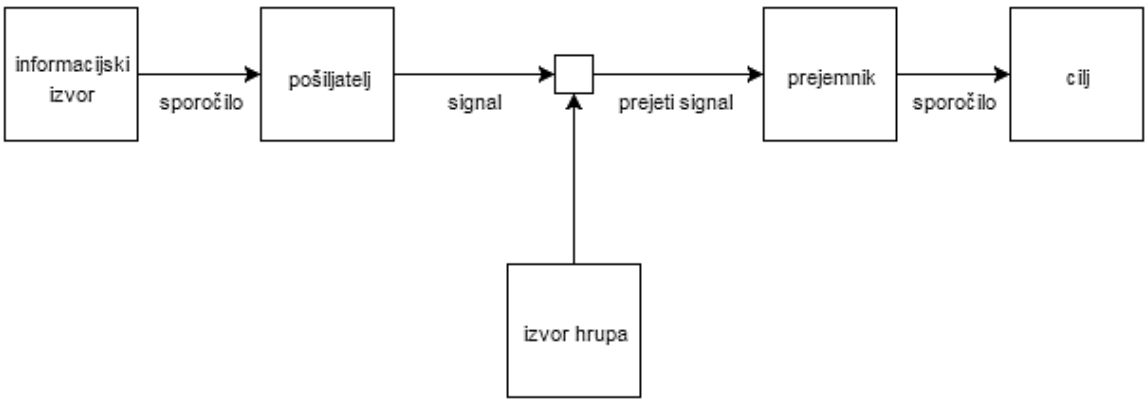
Izvemo, da za znak vsebuje 2 bita informacij in zato potrebujemo dvojiško kodo vsaj dolžine 2.

Informacijska teorija se ukvarja s pošiljanjem informacij prek komunikacijskega kanala od pošiljatelja do prejemnika. Pred informacijsko teorijo so za komunikacijo na daljavo uporabljali analogne signale. Za pošiljanje nekega sporočila je bilo treba spremeniti to sporočilo v različne impulze napetosti vzdolž žice, nakar smo izmerili napetost žice na drugem koncu in interpretirali impulze nazaj v besede. Tak način je zadoščal za

komuniciranje na krajše razdalje, a za daljše razdalje, npr. čez ocean, ni bilo mogoče. Z vsakim metrom kabla, po katerem mora analogni električni signal potovati, postaja šibkejši in je poleg tega izpostavljen naključnemu nihanju signala, ki mu rečemo hrup, zaradi materialov okoli žice.

Informacijska teorija je pomagala rešiti te probleme tako, da je definirala enote informacij, ki odražajo najmanjše možne dele, katerih se ne da zmanjšati. Za enote se najpogosteje uporablja bit, ki je na osnovi dvojiškega logaritma. S pomočjo teh enot lahko zakodiramo katero koli sporočilo.

Na podlagi te ideje lahko bistveno izboljšamo kvaliteto telekomunikacij. Sporočilo pretvorimo, črko za črko, v kode, sestavljene iz bitov `0` in `1`, ter pošljemo ta dolgi niz bitov po žici. Vsak bit `0` predstavlja kratek nizkonapetostni signal in vsak bit `1` pa kratek visokonapetostni signal. Tak signal je še zmeraj izpostavljen obema problemoma tako kot analogni signal, in sicer šibkost signala na daljše razdalje in hrup signala. Vendar ima digitalni signal prednost, ker je razmik med signali bitov `0` in `1` tako velik, da je kljub poslabšanju signala še vedno možno ponovno razbrati njihovo originalno stanje. Dodaten način ohranjanja čistega signala je, da na intervalih kabla uporabimo ponavljalnike, ki preberejo signal in posredujejo isti ter močnejši signal naprej. S tem je Shannon izumil novi moderni komunikacijski sistem, razviden iz Slika 2.4, ki se uporablja še dandanes [34][34].




Slika 2.4: Shannonova shema za moderni komunikacijski sistem.

Iz informacijske teorije vzamemo dva bistvena koncepta za boljše razumevanje uporabe kod VLC, to sta entropija in redundanca.

### 2.3.1 Entropija

Presenečanje oziroma informacija je uporabna za merjenje posameznih dogodkov oziroma branje znakov, vendar nas za doseganje kompresije bolj zanima količina informacij celotnega sporočila, kjer moramo upoštevati verjetnosti vseh znakov. Zato moramo pri informacijah znakov upoštevati tudi verjetnost dogodka. Entropijo izračunamo tako, da seštejemo vsa presenečanja posameznih znakov S = ($s_1$, $s_2$, ..., $s_n$) z verjetnostjo P = ($p_1$, $p_2$, ..., $p_n$) s formulo:

$$H(P) = \sum_{i=1}^{n} p_i I(s_i) = -\sum_{i=1}^{n} p_i \log_2 p_i \tag{2.6}$$

Označimo jo s črko H, poimenovano iz Boltzmannove H-teorije [5].

**Primer 2.7**: Imamo množico znakov S = (a, b, c) z verjetnostmi pojavitve P = ($\frac{1}{5}$, $\frac{2}{5}$, $\frac{2}{5}$). Izračunajmo entropijo.

$$\begin{aligned} H(P) &= -\sum_{i=1}^{n} p_i \log_2 p_i = -\left(\frac{1}{5} \times log_2 \frac{1}{5} + \frac{2}{5} \times log_2 \frac{2}{5} + \frac{2}{5} \times log_2 \frac{2}{5}\right) \\ &\approx -\big(-0{,}46 + (-0{,}53) + (-0{,}53)\big) \approx 1{,}52\ bitov/znak \end{aligned} \tag{2.7}$$

### 2.3.2 Redundanca

Kot je rečeno v Uvod, podatki običajno vsebujejo redundanco in lahko zato podatke stisnemo z odstranjevanjem te redundance. Za entropijo pravimo, da je pri svojem maksimumu, ko podatek nosi celotno informacijsko vsebino in se zaradi tega ne da več stisniti. Redundanco lahko torej definiramo kot količino, ki gre navzdol proti ničli, medtem ko se entropija bliža svojemu maksimumu. Izračunamo jo tako, da povprečni dolžini kod E (P, L), kjer so dolžine znakov označene z L = ($l_1$, $l_2$, ..., $l_n$) in verjetnosti znakov s P = ($p_1$, $p_2$, ..., $p_n$), odštejemo entropijo:

$$R(P, L) = E(P, L) - H(P) = \sum_{i=1}^{n} p_i l_i - (-\sum_{i=1}^{n} p_i log_2 p_i) \tag{2.8}$$

Redundanca je nič, kadar je povprečna dolžina kod enaka entropiji.

### 2.3.3 Optimalna koda

Kadar je redundanca nič, potem vemo, da je koda optimalna. V primerih, kadar redundanca ni nič, potem lahko s pomočjo entropije izračunamo optimalnost kode v odstotkih:

$$O(P,L) = \frac{H(P)}{R(P,L)} \times 100 \tag{2.9}$$

**Primer 2.8**: Imamo množico znakov S = (a, b, c, d) z distribucijo verjetnosti P = ($\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{8}$) in predponsko kodo C = (0, 10, 110, 111). Iz dolžin kod razberemo L = (1, 2, 3, 3). Izračunajmo optimalnost kod.

$$\begin{aligned} O(P,L) &= \frac{H(P)}{R(P,L)} \times 100 = \frac{-\sum_{i=1}^{4} p_i log_2 p_i}{\sum_{i=1}^{4} p_i l_i} \times 100 \\ &= \frac{-\left(\frac{1}{2} log_2 \frac{1}{2} + \frac{1}{4} log_2 \frac{1}{4} + \frac{1}{8} log_2 \frac{1}{8} + \frac{1}{8} log_2 \frac{1}{8}\right)}{\frac{1}{2} \times 1 + \frac{1}{4} \times 2 + \frac{1}{8} \times 3 + \frac{1}{8} \times 3} \times 100 \\ &= \frac{1{,}75}{1{,}75} \times 100 = 100\% \end{aligned} \tag{2.10}$$

**Primer 2.9**: Imamo podobno množico znakov S = (a, b, c, d), ampak tokrat z distribucijo verjetnosti P = ($\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{6}$, $\frac{1}{12}$) in predponsko kodo C = (0, 10, 110, 111). Iz dolžin kod ponovno razberemo L = (1, 2, 3, 3).

$$\begin{aligned} O(P,L) &= \frac{H(P)}{R(P,L)} \times 100 = \frac{-\sum_{i=1}^{4} p_i log_2 p_i}{\sum_{i=1}^{4} p_i l_i} \times 100 \\ &= \frac{-\left(\frac{1}{2} log_2 \frac{1}{2} + \frac{1}{4} log_2 \frac{1}{4} + \frac{1}{6} log_2 \frac{1}{6} + \frac{1}{12} log_2 \frac{1}{12}\right)}{\frac{1}{2} \times 1 + \frac{1}{4} \times 2 + \frac{1}{6} \times 3 + \frac{1}{12} \times 3} \times 100 \\ &= \frac{1{,}729}{1{,}75} \times 100 \approx 98{,}8\% \end{aligned} \tag{2.11}$$

Entropija določa zgolj teoretično mejo vrednosti informacij v podatkih, zato v praktičnih primerih težko dosežemo pridobitev optimalne kode.

# 3 Statistični algoritmi za stiskanje podatkov

Statistični algoritmi uporabljajo kode VLC, kjer krajše kode dodeljujemo bolj pogostim znakom oziroma skupini znakov. Glavna problema, na katera morajo implementacije statističnih algoritmov paziti, sta:

- dodeljevanje kod s predponsko lastnostjo (enolične in takojšnje kode),
- dodeljevanje kod z najmanjšimi povprečnimi dolžinami.

Prvi problem smo razjasnili v 2.2.3, medtem ko drugi problem rešuje vsak algoritem na svoj način.

## 3.1 Shannon-Fanojevo kodiranje

Ta algoritem je imenovan po Claudu E. Shannonu in Robertu M. Fanoju. Algoritem deluje tako, da ustvari množico kod VLC na podlagi znakov izvora in verjetnostne distribucije teh znakov. Metoda je delno optimalna v smislu, da ne doseže najkrajših možnih kodnih dolžin v primerjavi s Huffmanovim kodiranjem. Vendar pa za razliko od Huffmanovega kodiranja algoritem zagotavlja, da so vse dolžine kod VLC znotraj enega bita svojega teoretičnega ideala $-\log p_i$. Algoritem je sprva predlagal Shannon [16] leta 1948. Algoritem je kasneje leta 1949 dopolnil in pripisal Fano v svojem objavljenem tehničnem poročilu [8].

Algoritem začne z množico znakov z znano distribucijo verjetnosti. Znake sortiramo glede na njihove verjetnosti v padajočem vrstnem redu. Množico znakov razdelimo v dve podmnožici, tako da imata čim bolj podobni skupni verjetnosti. Nato vsem znakom ene podmnožice dodelimo začetno kodo `0`, znakom druge podmnožice pa dodelimo `1`. Vsako podmnožico nadalje rekurzivno razdeljujemo v njihove podmnožice približno enake verjetnosti in dodajamo kodam bite. Postopek traja, dokler podmnožice ne zaključimo z enim znakom.

**Primer 3.1**: Imamo množico znakov (`a`, `b`, `c`, `d`, `e`, `f`, `g`) z verjetnostmi (`0,25`; `0,20`; `0,15`; `0,15`; `0,10`; `0,10`; `0,05`). Znakom dodelimo kode VLC.

| Znak | Verjetnost | Koraki | | | | | | Koda VLC |
|---|---|---|---|---|---|---|---|---|
| a | 0,25 | 1 | 1 | | | | | 11 |
| b | 0,20 | 1 | 0 | | | | | 10 |
| c | 0,15 | 0 | | 1 | 1 | | | 011 |
| d | 0,15 | 0 | | 1 | 0 | | | 010 |
| e | 0,10 | 0 | | 0 | | 1 | | 001 |
| f | 0,10 | 0 | | 0 | | 0 | 1 | 0001 |
| g | 0,05 | 0 | | 0 | | 0 | 0 | 0000 |

Tabela 3.1: Postopek izgradnje kod VLC za Primer 3.1.

Iz Tabela 3.1 dobimo rezultat kodiranja: množica kod (`11`, `10`, `011`, `010`, `001`, `0001`, `0000`). V tabeli so delitve podmnožic prikazane s črto.

Da algoritem proizvede čim boljše kode s čim manjšimi dolžinami, morajo delitve množic na podmnožice biti čim učinkovitejše, kar pomeni, da morajo imeti podmnožice med seboj podobne verjetnosti.

**Primer 3.2**: Imamo množico znakov (`a`, `b`, `c`, `d`, `e`, `f`) z verjetnostmi (`0,25`; `0,25`; `0,125`; `0,125`; `0,125`; `0,125`). Znakom dodelimo kode VLC.

| Znak | Verjetnost | Koraki | | | | | Koda VLC |
|---|---|---|---|---|---|---|---|
| a | 0,25 | 1 | 1 | | | | 11 |
| b | 0,25 | 1 | 0 | | | | 10 |
| c | 0,125 | 0 | | 1 | 1 | | 011 |
| d | 0,125 | 0 | | 1 | 0 | | 010 |
| e | 0,125 | 0 | | 0 | | 1 | 001 |
| f | 0,125 | 0 | | 0 | | 0 | 000 |

Tabela 3.2: Postopek izgradnje kod VLC za Primer 3.2.

Iz Tabela 3.2 dobimo rezultat kodiranja: množica kod (`11`, `10`, `011`, `010`, `001`, `000`). V tabeli so delitve podmnožic ponovno prikazane s črto.

Iz Primer 3.2 ugotovimo, da algoritem proizvede optimalne kode z nič redundance, ampak to dosežemo samo takrat, kadar imajo vse delitve množic enake verjetnosti. Ali drugače, optimalne kode dobimo takrat, kadar imajo znaki distribucijo verjetnosti po formuli $2^{-x}$.

## 3.2 Huffmanovo kodiranje

Huffmanovo kodiranje je ena izmed najpopularnejših statističnih algoritmov za kodiranje znakov, ki ga je razvil David A. Huffman [10] leta 1952. Nekateri programi uporabljajo le Huffmanov algoritem za stiskanje podatkov, medtem ko ga ostali programi uporabljajo kot en korak v večkoračnem postopku.

Algoritem je podoben Shannon-Fanojevemu kodiranju, od katerega Huffmanovo kodiranje na splošno proizvede boljše kode. Enako kot Shannon-Fanojevo kodiranje, Huffmanovo kodiranje proizvede optimalne kode samo takrat, kadar verjetnosti znakov sledijo distribuciji $2^{-x}$. Glavna razlika med algoritmoma je ta, da Shannon-Fano gradi kodno drevo od zgoraj navzdol (kode gradi od leve proti desni), medtem ko Huffman gradi kodno drevo od spodaj navzgor (kode gradi od desne proti levi).

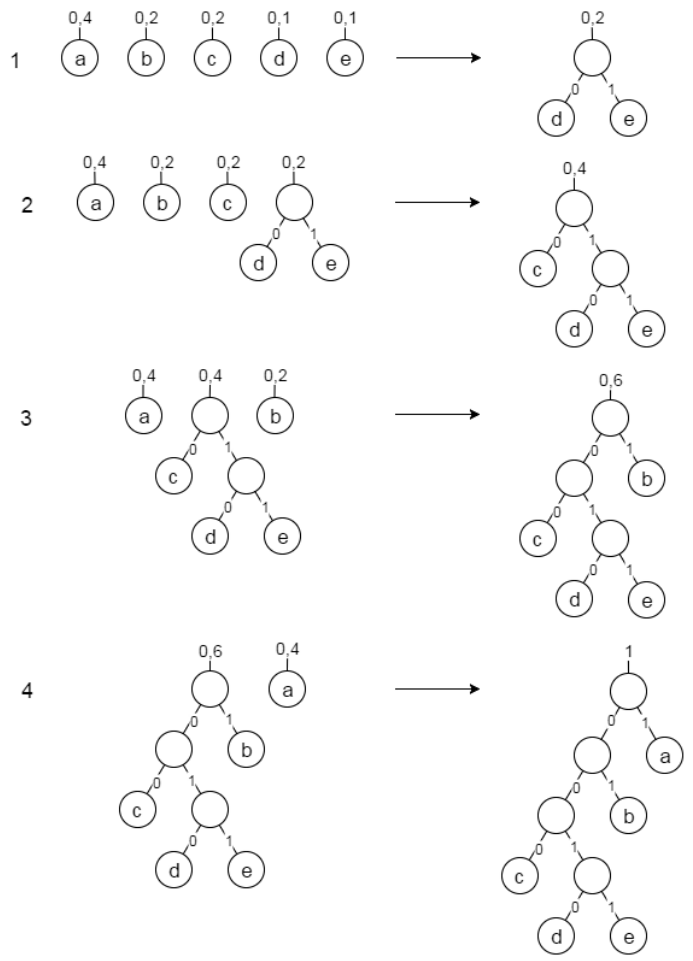
Robert G. Gallager je v svojem članku [9] dokazal, da je redundanca Huffmanovega kodiranja kvečjemu

$$p_i + \log_2 \frac{2 \log_2 e}{e} \approx p_i + 0{,}086\ bitov, \tag{3.1}$$

kjer $p_i$ predstavlja verjetnost nekega znaka abecede z največjo verjetnostjo.

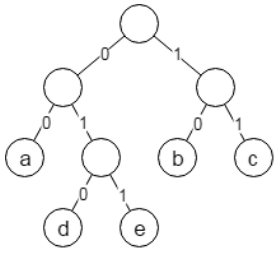
Za izgradnjo Huffmanovega drevesa začnemo tako, da ustvarimo seznam vseh znakov, sortiranih padajoče glede na njihovo distribucijo verjetnosti. Na začetku so vsi znaki predstavljeni kot listi drevesa. Iz seznama vzamemo dva znaka z najmanjšo verjetnostjo in ju združimo s pomožnim vozliščem, ki mu dodelimo seštevek verjetnosti združenih vozlišč, in vstavimo nazaj v seznam na pravo mesto, tako da seznam ostane sortiran. Leva povezava vozlišča npr. predstavlja bit `0`, desna povezava pa predstavlja bit `1`. Ta korak združevanja vozlišč ponavljamo, dokler nam ne ostane samo eno vozlišče v seznamu, ki predstavlja koren drevesa. Zdaj, ko imamo zgrajeno Huffmanovo drevo, nam preostane še sprehod po drevesu, da določimo znakom kode.

**Primer 3.3**: Imamo množico znakov (`a`, `b`, `c`, `d`, `e`) z verjetnostno distribucijo (`0,4`; `0,2`; `0,2`; `0,1`; `0,1`). Zgradimo Huffmanovo drevo.




Slika 3.1: Postopek izgradnje Huffmanovega drevesa za Primer 3.3.

S sprehodom po končnem drevesu iz Slika 3.1 dobimo množico kod $C_I$ = (`1`, `01`, `000`, `0010`, `0011`). Iz postopka opazimo, da smo lahko Huffmanovo drevo sestavili tudi drugače, ker smo imeli več kot dve vozlišči z enakimi verjetnostmi. To vidimo v Slika 3.1, kjer smo imeli vozlišče `b`, `c` in `de` z verjetnostjo `0,2`. Na spodnji Slika 3.2 je prikazano zgrajeno drevo, kadar izberemo drugačna vozlišča istih verjetnosti za isti Primer 3.3.




Slika 3.2: Huffmanovo drevo za Primer 3.3 z drugačno izbiro vozlišč.

Rezultat drevesa iz Slika 3.2 je množica drugačnih kod $C_2$ = (00, 10, 11, 010, 011). Čeprav so kode $C_1$ in $C_2$ različno dolge, vidimo na spodnjih izračunih, da povprečna dolžina obeh kod ostaja enaka.

$C_1$:
$$E(P,L) = \sum_{i=1}^{n} p_i l_i = 0{,}4 \times 1 + 0{,}2 \times 2 + 0{,}2 \times 3 + 0{,}1 \times 4 + 0{,}1 \times 4 = 2{,}2\ bitov/znak \tag{3.2}$$

$C_2$:
$$E(P,L) = \sum_{i=1}^{n} p_i l_i = 0{,}4 \times 2 + 0{,}2 \times 2 + 0{,}2 \times 2 + 0{,}1 \times 3 + 0{,}1 \times 3 = 2{,}2\ bitov/znak \tag{3.3}$$

A vseeno se izkaže, da kodi obeh primerov nista povsem enakovredni, kajti razlikujeta se po varianci. Varianca neke kode nam pove, koliko se velikost posameznih kod razlikuje od skupne povprečne velikosti. Kode z manjšo varianco so bolj priporočene, izračunamo jo s formulo:

$$V = \sigma^2 = \frac{1}{N}\sum (X - \mu)^2 \tag{3.4}$$

S pomočjo formule lahko izračunamo varianco prejšnjih kod $C_1$ in $C_2$:

$C_1$:
$$V = \sum_{i=1}^{5} p_i (l_i - E(P,L))^2 = 0{,}4 \times (1 - 2{,}2)^2 + 0{,}2 \times (2 - 2{,}2)^2 + 0{,}2 \times (3 - 2{,}2)^2 + 0{,}1 \times (4 - 2{,}2)^2 + 0{,}1 \times (4 - 2{,}2)^2 = 1{,}36 \tag{3.5}$$

$C_2$:
$$V = \sum_{i=1}^{5} p_i (l_i - E(P,L))^2 = 0{,}4 \times (2 - 2{,}2)^2 + 0{,}2 \times (2 - 2{,}2)^2 + 0{,}2 \times (2 - 2{,}2)^2 + 0{,}1 \times (3 - 2{,}2)^2 + 0{,}1 \times (3 - 2{,}2)^2 = 0{,}16 \tag{3.6}$$

Ker vidimo iz izračuna (3.6), da imajo kode $C_2$ manjšo varianco od $C_1$, so le-te bolj zaželene. Da poskrbimo za čim manjšo varianco kod, velja pravilo: ko imamo tri ali več vozlišč z istimi najmanjšimi verjetnostmi, izberemo tisti dve vozlišči, tako da poskrbimo, da so znaki z majhno verjetnostjo združeni z znaki velike verjetnosti.

Če kodirnik pošilja stisnjen izhodni tok le v datoteko, potem varianca ni pomembna. Majhna varianca je zaželena samo takrat, ko kodirnik pošilja stisnjen izhodni tok istočasno, kot se generira. V takem primeru koda z visoko varianco povzroča, da kodirnik generira bite s hitrostjo, ki niha ves čas. In ker hočemo pošiljati bite v izhodni tok s konstantno hitrostjo, moramo pri visoki varianci uporabiti večji medpomnilnik v primerjavi z majhno varianco.

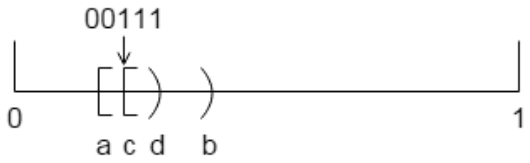
## 3.3 Aritmetično kodiranje

Huffmanovo kodiranje je preprost in učinkovit algoritem, ki proizvede najboljše kode za posamezne znake. Ampak, kot smo že omenili v prejšnjem 15, proizvaja optimalne kode VLC samo takrat, kadar vse verjetnosti pojavitve znakov sledijo formuli $2^{-x}$. Do tega pride zato, ker Huffmanova koda dodeljuje vsem znakom abecede samo celoštevilske bite. Informacijska teorija nam pove, da bi za znak z verjetnostjo `0,4` idealno morali dodelili 1,32-bitno kodo, kajti informacija tega znaka znaša $-log_2 0{,}4 \approx 1{,}32$. V tem primeru je Huffmanova koda zmožna znaku dodeliti samo enobitno ali dvobitno kodo in s tem bi dobili redundanco. Aritmetično kodiranje se izogne temu primeru tako, da namesto dodeljevanja kod vsakemu posameznemu znaku dodeli eno dolgo kodo celotnemu izvoru oziroma vhodni datoteki.

Princip aritmetičnega kodiranja je sprva predlagal Peter Elias nekje pred letom 1963 v svojem neobjavljenem delu. Prve implementacije so razvili Jorma J. Rissanen leta 1976, Richard C. Pasco leta 1976 in Frank Rubin leta 1979. Za razliko od Shannon-Fanojevega in Huffmanovega kodiranja, ima aritmetično kodiranje ločena postopka za statistično modeliranje in kodiranje podatkov. Prav tako ima prednost, da za statistični model lahko sprejme kateri koli model, tudi če verjetnosti niso neodvisne in identično porazdeljene (angl. independent and identically distributed, IID), in ohrani učinkovitost algoritma.

Metoda začne z modeliranjem podatkov, tako da vzame začetni interval [0,1) in ga razdeli na podintervale, sorazmerno velike z verjetnostmi pojavitve vseh znakov, kjer vsak del intervala predstavlja nek znak. Ko ima razdeljene podintervale, začne z branjem posameznih znakov vhodnega toka in izbere tisti podinterval, ki mu pripada prebrani znak. Ta podinterval zopet razdeli na podintervale, sorazmerno velike z verjetnostmi vseh znakov. Ta postopek delitve podintervalov ponavljamo tolikokrat, dokler ne preberemo celotnega vhodnega toka. Rezultat postopka modeliranja je nek podinterval [a, b) v intervalu [0, 1).

V drugem delu postopka kodiranja pa metoda poišče najkrajšo dvojiško število, ki jo predstavlja največji podinterval [c, d) v iskanem intervalu [a, b) iz prejšnjega postopka. To storimo tako, da ponovno začnemo z novim intervalom [0, 1), toda ga tokrat razdelimo na dvojiške velikosti oziroma polovice, kjer npr. spodnja polovica predstavlja bit `0` in zgornja polovica bit `1`. Glede na to, kje se nahaja iskani interval [a, b), izberemo ustrezen trenutni podinterval in ponovno razdelimo na polovice. Postopek ponavljamo, dokler nimamo intervala, ki je vsebovan znotraj [a, b). Končno dvojiško število predstavlja kodo celotnega sporočila. Na Slika 3.3 je prikazan končni združeni interval [0, 1), kjer se koda sporočila nahaja v točki c.




Slika 3.3: Ponazoritev aritmetičnega kodiranja.

Za dekodiranje storimo podobno, toda obratno. Najprej s pomočjo zakodiranega sporočila naredimo bisekcije intervala [0, 1), po katerem se sprehajamo na podlagi bitov kode, dokler ne pridemo do zadnjega podintervala, ki predstavlja interval [c, d).

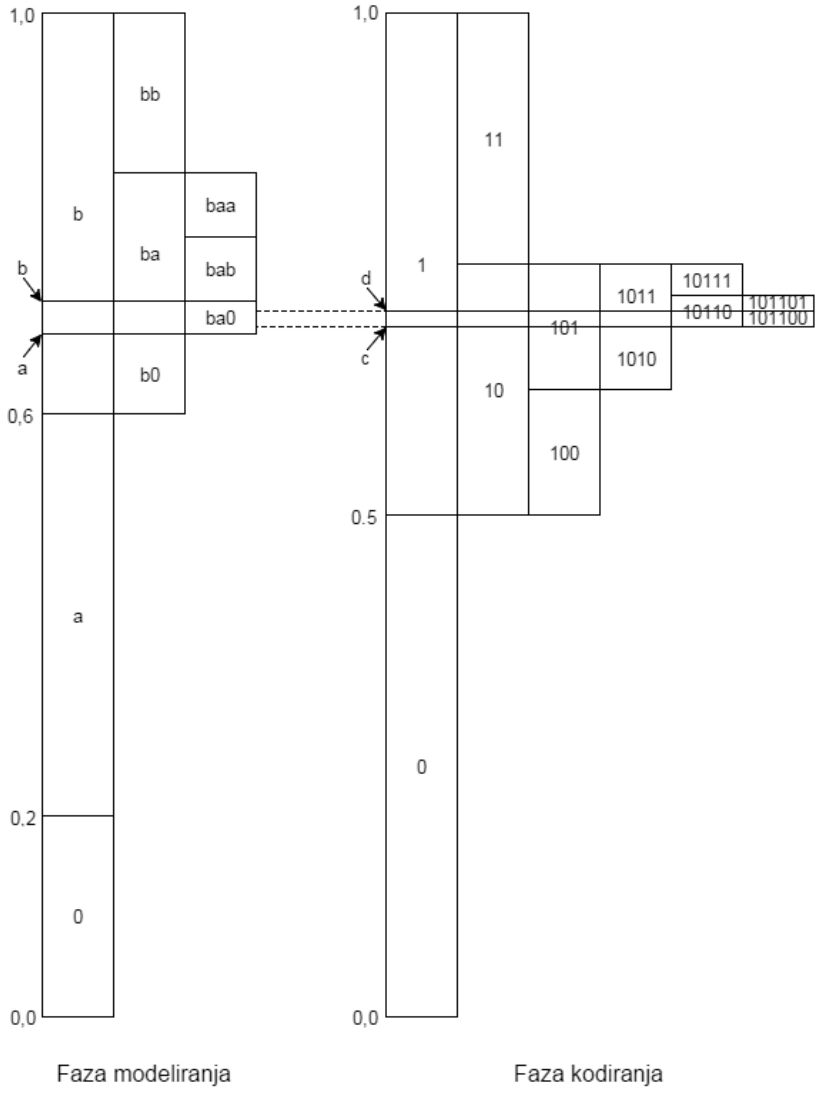
Dekodiranje nadaljujemo z razdelitvijo novega intervala [0, 1) na sorazmerne velikosti verjetnostne distribucije znakov in tukaj izbiramo tiste podintervale, ki vsebujejo interval [c, d). Z vsakim izbranim podintervalom točno vemo, kateri znak je predstavljen v tem delu intervala.

Pri drugem delu zgornje razlage dekodiranja lahko nastane problem, saj ne vemo, kdaj moramo nehati z delitvijo drugega intervala. Pri kodiranju moramo zato storiti še eno od naslednjega:

- dodatno zakodiramo dolžino originalnega sporočila,
- uporabimo znak EOF.

V praksi se uporablja znak EOF.

**Primer 3.4**: Imamo množico znakov (`0`, `a`, `b`) z verjetnostno distribucijo (`0,2`; `0,4`; `0,4`). Znak `0` predstavlja znak EOF. Poskušamo zakodirati sporočilo `ba0`.




Slika 3.4: Postopek kodiranja za Primer 3.4.

Iz Slika 3.4 vidimo, da je končni rezultat postopka koda `101100`. Da se zagotovo prepričamo, ali leži interval [c, d) v intervalu [a, b), jih izračunamo s pomočjo seštevka višin spodnjih intervalov:

$$[a, b) = [0{,}2 + 0{,}4 + 0{,}4 \times 0{,}2;\ a + 0{,}4 \times 0{,}4 \times 0{,}2) = [0{,}68;\ 0{,}68 + 0{,}032) = [0{,}68;\ 0{,}712) \tag{3.7}$$

$$[c, d) = [0{,}5 + 0{,}125 + 0{,}0625;\ c + 0{,}015625) = [0{,}6875;\ 0{,}6875 + 0{,}015625) = [0{,}6875;\ 0{,}703125) \tag{3.8}$$

Iz izračunov vidimo, da je interval [c, d) popolnoma vsebovan v intervalu [a, b).

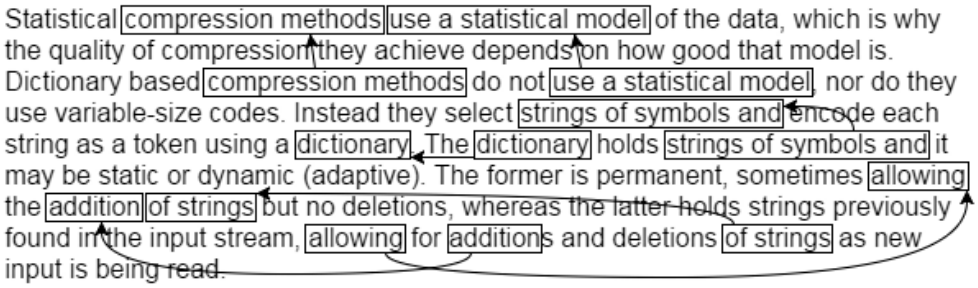
Do zdaj smo opisali poenostavljeno različico algoritma, ker uporablja neskončno natančnost decimalnih števil. Ker računalniki računajo s končnim številom decimalk, se problemu izognemo tako, da ob vsaki izbiri podintervala razširimo ta interval z dvakratnim množenjem njegovih mej. Zaradi postopka razširjanja dobimo nekaj dodatne redundance, toda ta količina je zanemarljiva in koda je še zmeraj zelo v bližini optimalnosti.

Razlog, zakaj moramo poiskati dvojiško število, katerega interval [c, d) je popolnoma vsebovan v intervalu [a, b), je ta, da poskrbimo za možnost enoličnega dekodiranja kode. To pa potrebujemo samo v primeru, če združujemo več zakodiranih sporočil skupaj. Če ne združujemo, potem lahko uporabimo krajšo predstavitev dvojiškega intervala. Za naš prejšnji Primer 3.4 bi torej lahko nadomestili kodo `101100` s kodo `1011`.

Kljub številnim prednostim se aritmetično kodiranje ne uporablja veliko v praksi. Razlogi za to so bili razni patenti v preteklosti. Zaradi strahu pred kršitvijo patentov in plačevanja licenc, kot je bil to primer s podjetjem Unisysom, ki je v 4.4 podrobneje opisano, skoraj nobena programska oprema ni implementirala aritmetičnega kodiranja. Program Bzip2 je sprva uporabljal aritmetično kodiranje, a ga je kasneje nadomestil s Huffmanovim kodiranjem ravno zaradi prejšnjih razlogov. Še en velik primer je format JPEG, ki dovoljuje uporabo obeh Huffmanovega in aritmetičnega kodiranja, a se v praksi uporablja le Huffmanovo kodiranje. Kljub temu da so vsi patenti že potekli [23], [24], [25], se uporaba aritmetičnega kodiranja še zmeraj ne povečuje zaradi vztrajnosti programskih oprem in težavnosti spreminjanja že obstoječih ustanovljenih formatov.

# 4 Algoritmi za stiskanje podatkov s slovarji

Statistični algoritmi stiskanja podatkov, kot sta Huffmanovo kodiranje in aritmetično kodiranje, uporabljajo statistični model podatkov, kar je tudi razlog, da je kvaliteta stiskanja podatkov odvisna od velikosti izvora in koliko je podobna verjetnostna distribucija od samega statističnega modela. Tudi njihovi algoritmi vplivajo na učinkovitost stiskanja, saj morajo pri kodah VLC njihove dolžine zadoščati Kraftovi neenačbi, če želijo biti enolične in takojšnje. To pa predstavlja omejitev zmogljivosti teh algoritmov.




Slika 4.1: Ponavljajoči se nizi v tekstovnem besedilu.

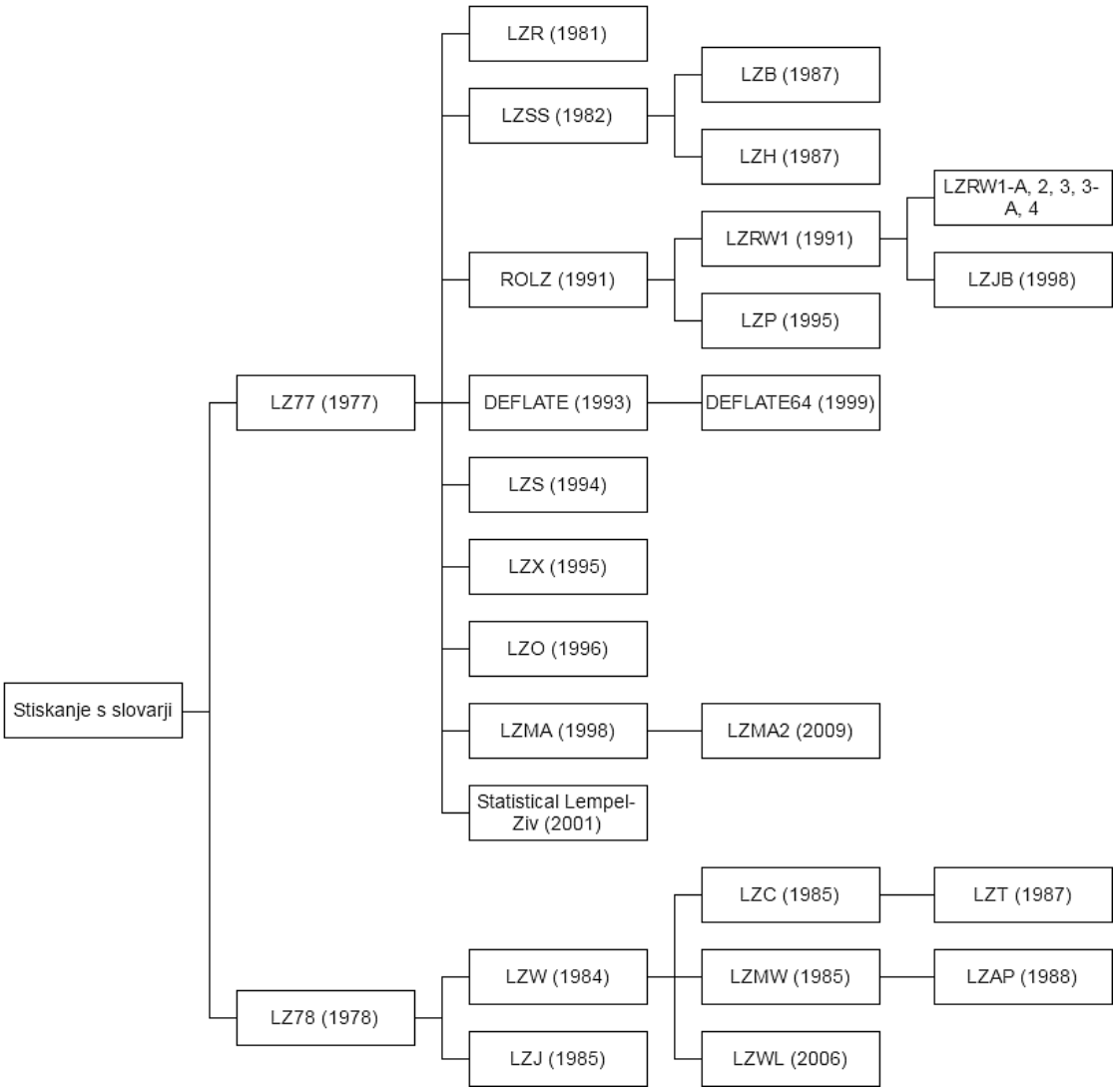
Običajno se največji vir redundance podatkov kaže v obliki ponavljajočih se nizov besedila. Tak tip redundance vidimo v vseh oblikah podatkov – avdio, video, slikah in predvsem v tekstovnem besedilu. Slika 4.1 prikazuje tak primer redundance.

Algoritmi za stiskanje podatkov s pomočjo slovarja ne uporabljajo statističnega modela. Cilj algoritmov je, da se znebijo redundance s shranjevanjem ponavljajočih se nizov v slovarju in te nize zakodirajo v obliki žetonov. Žeton nosi informacijo, s katero lahko enolično identificiramo najden niz v slovarju. Struktura žetona se med posameznimi algoritmi razlikuje. Žetoni so v najboljšem primeru precej krajši od samega niza, ki ga žeton nadomešča. Zakodiran tok je po navadi sestavljen iz mešanice žetonov in nestisnjenih znakov.

Glavne operacije algoritmov s slovarji so primerjava nizov, vzdrževanje slovarja in kodiranje na učinkovit način. Taki algoritmi so po naravi adaptivni, ker se slovar neprestano posodablja med procesom kompresije in dekompresije. Vsebina slovarja se spreminja odvisno od podatkov iz vhodnega toka. Kodirnik in dekodirnik vzdržujeta svoj lasten slovar.

Algoritmi s slovarji ne uporabljajo statističnega modela podatkov, temveč se zanašajo na prepoznavanje ponavljajočih se nizov. Tako da učinkovitost stiskanja podatkov ni odvisna od kvalitete statističnega modela in prav tako ni omejena z entropijo izvora. Zato lahko pogosto dosežemo boljše razmerje stiskanja kot pa prejšnji algoritmi s statističnim modelom.

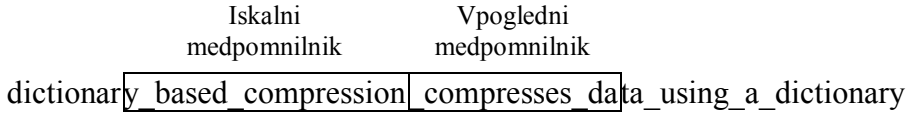
Prve algoritme s slovarji sta razvila Jacob Ziv in Abraham Lempel leta 1977 [20] in 1978 [21]. Njuna prvotna algoritma LZ77 in LZ78 so kasneje mnogi drugi raziskovalci izboljšali in združevali z drugimi algoritmi in tako je nastala velika družina algoritmov Lempel-Ziv. Slika 4.2 prikazuje najbolj popularne in uporabljene algoritme Lempel-Ziv, vendar je poleg teh še mnogo več variant in jih je težko vse prikazati [30].




Slika 4.2: Hierarhija algoritmov Lempel-Ziv.

## 4.1 LZ77

Princip algoritma LZ77 (znan tudi kot LZ1) je, da uporablja del že videnega vhodnega toka kot slovar. Ta slovar ni izrecno zunanji slovar, vendar je t. i. drseče okno, sestavljeno iz dveh delov. Prvi oziroma levi del okna se imenuje iskalni medpomnilnik (angl. search buffer), ki vsebuje že obdelane znake. Drugi del oziroma desni del okna se imenuje vpogledni medpomnilnik (look-ahead buffer), ki vsebuje znake, ki jih je treba še zakodirati. V praktičnih implementacijah je velikost iskalnega medpomnilnika nekaj tisoč znakov (običajno 4096 znakov = 12 bitov), medtem ko je vpogledni medpomnilnik velik le nekaj desetin znakov (običajno 16 znakov = 4 bitov).




Slika 4.3: Primer drsečega okna.

Algoritem LZ77 vzvratno poišče (od desne proti levi) v iskalnem medpomnilniku najdaljše ujemanje niza z začetnim nizom vpoglednega medpomnilnika. V zgornji Slika 4.3 bi kodirnik začel z iskanjem prvega znaka _, ki predstavlja znak presledka, iz vpoglednega medpomnilnika. V iskalnem medpomnilniku najdemo znak presledka na razdalji 12, gledano od konca iskalnega medpomnilnika. Med obema znakoma presledka določimo najdaljši ujemajoči niz. V tem primeru je to 9 znakov, ki predstavlja niza `_compress`. Kodirnik nato nadaljuje z vzvratnim iskanjem in poskuša najti novi ujemajoči niz, daljšega od dolžine 9. V našem primeru najdemo še en znak presledka na razdalji 18, a se pri njem ujema samo z enim znakom. Ko kodirnik konča z iskanjem, izbere najdaljši najden niz in ga zakodira v obliki žetona (`12,9,e`).

Žetoni algoritma LZ77 so sestavljeni iz treh delov: razdalje, dolžine in naslednjega znaka od najdenega niza v vpoglednem medpomnilniku. V žetonu je podan znak `e`, kajti to je naslednji znak najdenega niza v besedi `compresses` iz Slika 4.3. Razlog za vključitev naslednjega znaka niza je za tiste situacije, kadar iskalni medpomnilnik ne vsebuje iskanega niza. Takrat podamo žeton z razdaljo in dolžino 0 ter za znak ostane kar prvi znak vpoglednega medpomnilnika. Žetone zapišemo v izhodni tok in istočasno premaknemo položaj okna na desno za toliko mest, kolikor je dolžina najdenega niza plus eno mesto za naslednji znak. V zgornjem primeru bi premaknili za 10 mest; 9 mest zaradi dolžine niza `_compress` in 1 mesto zaradi znaka `e`.

**Primer 4.1**: Imamo drseče okno z iskalnim medpomnilnikom velikosti 8 znakov in vpoglednim medpomnilnikom velikosti 8 znakov. Poskušamo zakodirati niz `aacaacabcababaddddc`.

| Korak | Iskalni medpomnilnik | Vpogledni medpomnilnik | Izhodni žeton |
|---|---|---|---|
| 1 | | aacaacab | (0,0,a) |
| 2 | a | acaacabc | (1,1,c) |
| 3 | aac | aacabcab | (3,4,b) |
| 4 | aacaacab | cabacadd | (3,3,a) |
| 5 | acabcaba | caddddc | (7,2,d) |
| 6 | bcabacad | dddc | (1,3,c) |

Tabela 4.1: Postopek kodiranja za Primer 4.1.

Iz Tabela 4.1 dobimo rezultat kodiranja, množico žetonov (`0`,`0`,`a`), (`1`,`1`,`c`), (`3`,`4`,`b`), (`3`,`3`,`a`), (`7`,`2`,`d`), (`7`,`2`,`d`). Če upoštevamo, da znaki zasedajo 8 bitov in razdalja ter dolžina vsaka po 3 bite, dobimo velikost zakodiranega sporočila: $(3 + 3 + 8) \times 6 = 84$ bitov, kar je skoraj dvakrat manj v primerjavi z nezakodiranem sporočilom: $8 \times 9 = 152$ bitov.

Če kodirnik pri iskanju najdaljšega niza naleti na več enako dolgih nizov, potem običajno izbere zadnji niz, saj mora s tem spremljati le zadnji viden ujemajoči niz, kar poenostavi implementacijo. Tudi mi smo imeli tak primer pri Tabela 4.1 z nizom `ca` na razdalji 4 in 7. Čeprav je pri nekaterih različicah algoritma LZ77 bolje izbrati niz z nižjo razdaljo za doseganje boljše učinkovitosti. Ena takšna različica je algoritem LZH, ki združi LZ77 s Huffmanovim kodiranjem, ki mu koristi imeti čim krajše razdalje zaradi krajših kod VLC.

Pri določanju dolžine najdenega niza je možno imeti tudi večjo dolžino od razdalje, a posledica tega je, da niz preide iz iskalnega medpomnilnika v vpogledni medpomnilnik. To opazimo pri Tabela 4.1. Tak način kodiranja ne predstavlja problemov, ker lahko dekodirnik obravnava take žetone enostavno brez kakršnih koli modifikacij.

Še ena uporabna lastnost algoritma LZ77 je ta, da lahko z žetonom predstavimo večkrat ponavljajoči znak ali niz. Tak primer smo imeli pri zadnjem Tabela 4.1, kjer smo znak `d` trikrat ponovili. Se pravi efektivno je možno z LZ77 posnemati delovanje algoritma za stiskanje RLE.

Dekodiranje je enostavnejše od kodiranja. Dekodirnik mora prav tako vzdrževati svoj medpomnilnik, ampak tokrat samo enega velikosti iskalnega medpomnilnika. Dekodirnik prebere žeton iz vhodnega toka, poišče iskani niz v medpomnilniku in zapisuje znake v

izhodni tok, dokler ne pride do konca niza, medtem ko pri vsakem znaku premika celotni medpomnilnik.

**Primer 4.2**: Imamo medpomnilnik velikosti 8 znakov. Poskušamo dekodirati žetone (0,0,a), (1,1,c), (3,4,b), (3,3,a), (7,2,d), (7,2,d) iz Primer 4.1.

| Korak | Vhodni žeton | Trenutni medpomnilnik | Izhodni znak | Posodobljeni medpomnilnik |
|---|---|---|---|---|
| 1 | (0,0,a) | | a | a |
| 2a | (1,1,c) | a | a | aa |
| 2b | (1,0,c) | aa | c | aac |
| 3a | (3,4,b) | aac | a | aaca |
| 3b | (3,3,b) | aaca | a | aacaa |
| 3c | (3,2,b) | aacaa | c | aacaac |
| 3d | (3,1,b) | aacaac | a | aacaaca |
| 3e | (3,0,b) | aacaaca | b | aacaacab |
| 4a | (3,3,a) | aacaacab | c | acaacabc |
| 4b | (3,2,a) | acaacabc | a | caacabca |
| 4c | (3,1,a) | caacabca | b | aacabcab |
| 4d | (3,0,a) | aacabcab | a | acabcaba |
| 5a | (7,2,d) | acabcaba | c | cabcabac |
| 5b | (7,1,d) | cabcabac | a | abcabaca |
| 5c | (7,0,d) | abcabaca | d | bcabacad |
| 6a | (1,3,c) | bcabacad | d | cabacadd |
| 6b | (1,2,c) | cabacadd | d | abacaddd |
| 6c | (1,1,c) | abacaddd | d | bacadddd |
| 6d | (1,0,c) | bacadddd | c | acaddddc |

Tabela 4.2: Postopek dekodiranja za Primer 4.2.

Iz Tabela 4.2 dobimo rezultat dekodiranja, niz `aacaacabcabacaddddc`, ki se ujema z originalnem nizom pred kodiranjem.

Kadar ima žeton dolžino 0, potem enostavno zapišemo znak iz žetona v izhodni tok in posodobimo medpomnilnik. A kadar ima dolžino večjo od 0, smo razdelili dani korak na več manjših korakov, v katerih smo zapisovali posamezne znake. Za pomoč spremljanja teh manjših korakov smo zmanjševali dolžino žetona, dokler nismo dobili dolžine 0.

Dekodiranje je precej hitrejše od kodiranja, zato se algoritem LZ77 in njegove različice običajno uporabljajo v situacijah, kjer se podatki le enkrat stisnejo in zelo pogosto razširjajo. En tak primer je uporaba različice Deflate pri HTTP-stiskanju.

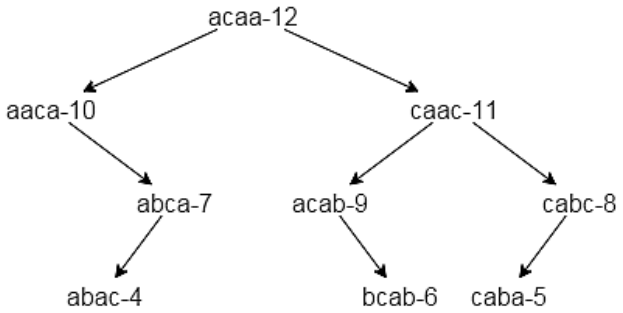
Ena tipična izboljšava algoritma LZ77 in njegovih različic je, da za iskalni medpomnilnik uporabimo dvojiško iskalno drevo. Za primerjavo med vozlišči uporabimo kar vrednosti kod

ASCII, a ker imamo niz z več znaki, jih primerjamo glede na njihov leksikografski vrstni red. V dvojiško drevo vstavimo vse mogoče nize iz iskalnega medpomnilnika skupaj z njihovimi razdaljami. Dolžina niza je enaka velikosti vpoglednega medpomnilnika.

**Primer 4.3**: Imamo trenutno drseče okno `acaacabcabac` `addd`. Iskalni medpomnilnik nadomestimo z dvojiškim iskalnim drevesom.

| Niz | Razdalja |
|---|---|
| acaa | 12 |
| caac | 11 |
| aaca | 10 |
| acab | 9 |
| cabc | 8 |
| abca | 7 |
| bcab | 6 |
| caba | 5 |
| abac | 4 |

Tabela 4.3: Prikaz vseh nizov dolžine 4 z razdaljami za Primer 4.3.




Slika 4.4: Dvojiško iskalno drevo za Primer 4.3.

Tabela 4.3 vsebuje vse kombinacije nizov dolžine 4 iz iskalnega medpomnilnika skupaj z njihovimi razdaljami. Iz teh nizov smo nato zgradili dvojiško iskalno drevo, ki je prikazano na Slika 4.4. Ker se dvojiško drevo neprestano posodablja, na isti način kot drseče okno, ima omejeno velikost, ki jo lahko izračunamo s pomočjo iskalnega medpomnilnika I in vpoglednega medpomnilnika V. Za naš prejšnji Primer 4.3 bi izračunali kot: $I - V + 1 = 12 - 4 + 1 = 9$ vozlišč, kar se tudi ujema z našim drevesom.

## 4.2 LZSS

Glavna pomanjkljivost algoritma LZ77 se pokaže pri kodiranju niza, kadar le-ta ni najden v slovarju. LZ77 vedno izpiše žetone v izhodni tok, tudi takrat, kadar je žeton večji od samega

niza. To pomanjkljivost sta rešila James Storer in Thomas Szymanski leta 1982 [17] z njunim algoritmom LZSS.

Algoritem LZSS se od svojega predhodnika razlikuje v načinu zapisovanja izhodnega toka. Namesto zapisovanja samo tridelnih žetonov (razdalja, dolžina, znak), LZSS zapiše dvodelne žetone (razdalja, dolžina), toda samo takrat, kadar je žeton manjši od najdenega niza. V primeru, da je žeton večji od niza, potem v izhodni tok zapišemo prvi nezakodiran znak iz vpoglednega medpomnilnika. Da lahko dekodirnik razlikuje med znaki in žetoni, mora kodirnik pred vsakim zapisom dodati enobitno zastavico z namenom za ločevanje med znaki in žetoni. Zastavica `0` lahko pomeni, da gre za nestisnjen znak, zastavica `1`, da gre za žeton.

**Primer 4.4**: Imamo drseče okno z iskalnim medpomnilnikom velikosti 8 znakov in vpoglednim medpomnilnikom velikosti 8 znakov. Poskušamo zakodirati niz `aacaacabcabacaddddc`.

| Korak | Iskalni medpomnilnik | Vpogledni medpomnilnik | Izhodni žeton |
|---|---|---|---|
| 1 | | aacaacab | 0a |
| 2 | a | acaacabc | 1(1,1) |
| 3 | aa | caacabca | 0c |
| 4 | aac | aacabcab | 1(3,4) |
| 5 | aacaaca | bcabacad | 0b |
| 6 | aacaacab | cabacadd | 1(3,3) |
| 7 | aacabcab | acaddddc | 1(7,3) |
| 8 | abcabaca | ddddc | 0d |
| 9 | bcabacad | dddc | 1(1,3) |
| 10 | bacadddd | c | 1(6,1) |

Tabela 4.4: Postopek kodiranja za Primer 4.4.

Iz Tabela 4.4 dobimo rezultat kodiranja, izhodni tok `0a`, `1(1,1)`, `0c`, `1(3,4)`, `0b`, `1(3,3)`, `1(7,3)`, `0d`, `1(1,3)`, `1(6,1)`. Če ponovno upoštevamo, da so znaki veliki 8 bitov in vsak medpomnilnik 3 bite ter 1 bit za zastavice, izračunamo velikost zakodiranega sporočila: $4 \times (1 + 8) + 6 \times (1 + 3 + 3) = 78$ bitov. Dobili smo boljše razmerje stiskanja kakor pri LZ77, kjer smo dobili 84 bitov.

Ker smo v našem primeru uporabili relativno majhna medpomnilnika, skupaj 6 bitov, se nam je pri Tabela 4.4 bolj izplačalo zapisati žeton kot pa sam znak. V praksi sta seveda medpomnilnika večja, najverjetneje 16 bitov skupaj, zato bi po navadi potrebovali niz z vsaj dolžino 3, da zapišemo žeton namesto znaka.

**Primer 4.5**: Imamo medpomnilnik velikosti 8 znakov. Poskušamo dekodirati vhodni tok `0a`, `1(1,1)`, `0c`, `1(3,4)`, `0b`, `1(3,3)`, `1(7,3)`, `0d`, `1(1,3)`, `1(6,1)` iz Primer 4.4.

| Korak | Vhodni tok | Trenutni medpomnilnik | Izhodni znak | Posodobljeni medpomnilnik |
|---|---|---|---|---|
| 1 | 0a | | a | a |
| 2 | 1(1,1) | a | a | aa |
| 3 | 0c | aa | c | aac |
| 4a | 1(3,4) | aac | a | aaca |
| 4b | 1(3,3) | aaca | a | aacaa |
| 4c | 1(3,2) | aacaa | c | aacaac |
| 4d | 1(3,1) | aacaac | a | aacaaca |
| 5 | 0b | aacaaca | b | aacaacab |
| 6a | 1(3,3) | aacaacab | c | acaacabc |
| 6b | 1(3,2) | acaacabc | a | caacabca |
| 6c | 1(3,1) | caacabca | b | aacabcab |
| 7a | 1(7,3) | aacabcab | a | acabcaba |
| 7b | 1(7,2) | acabcaba | c | cabcabac |
| 7c | 1(7,1) | cabcabac | a | abcabaca |
| 8 | 0d | abcabaca | d | bcabacad |
| 9a | 1(1,3) | bcabacad | d | cabacadd |
| 9b | 1(1,2) | cabacadd | d | abacaddd |
| 9c | 1(1,1) | abacaddd | d | bacadddd |
| 10 | 1(6,1) | bacadddd | c | acaddddc |

Tabela 4.5: Postopek dekodiranja za Primer 4.5.

Iz Tabela 4.5 dobimo rezultat dekodiranja niz `aacaacabcabacaddddc`, ki se ujema z originalnim nizom.

Poleg izboljšav LZ77 lahko algoritem LZSS izboljšamo tudi tako, da grupiramo osem zastavic skupaj in nato osem znakov ali žetonov, odvisno od vrstnega reda zastavic. Namesto stisnjenega toka

`0k, 1(165,7), 1(389,13), 0s, 0t, 1(81,4), 1(47,10), 1(96,5)`

bi zapisali kot

`01100111, k, (165,7), (389,13), s, t, (81,4), (47,10), (96,5).`

Namen tega je, da dekodirniku zmanjšamo potrebo po bitnih operacijah, saj dobimo s pomočjo grupiranja vnose samo velikosti bajtov in ne rabimo brati bit za bitom [32].

## 4.3 LZ78

Slabost algoritma LZ77 in večine njegovih različic se kaže v velikosti slovarja oziroma drsečega okna, ker zna biti zelo majhno od izvora podatkov. Ti algoritmi lahko zato primerjajo trenutne podatke le z majhnim delom že videnih podatkov. Da bi se izognili temu problemu, bi lahko povečali velikost drsečega okna, toda s tem bi posledično tudi povečali velikost žetonov. Večji žetoni imajo v večini primerov negativen učinek na učinkovitost stiskanja podatkov. Zaradi večjih žetonov se tudi poveča minimalna dolžina ujemanja znakov, npr. iz treh znakov na štiri, pri čemer dobimo več odvečnih enobitnih zastavic. A glavna posledica večjega drsečega okna je drastično povečanje procesorskega časa, zaradi česar se precej poveča čas kodiranja [12].

Da bi se izognila temu problemu, sta Ziv in Lempel razvila algoritem LZ78 (znan tudi kot LZ2). Ta ne uporablja drsečega okna, ampak dejanski zunanji slovar, v katerem shranjuje pretekle že videne nize. Algoritem začne s praznim slovarjem, velikost katerega je teoretično omejena le s količino prostega pomnilnika. LZ78 se od LZ77 razlikuje tudi v izhodnih žetonih. Tukaj so žetoni sestavljeni iz dveh delov (indeks, znak). Prvi del žetona predstavlja indeks slovarja, drugi del pa predstavlja naslednji nezakodiran znak. Žeton ne vsebuje podatka o dolžini niza, kajti niz razberemo iz slovarja, zaradi česar potrebujemo njegov indeks. Iz slovarja se nič ne briše, temveč se samo dodajajo novi vnosi. To je prednost pred algoritmom LZ77, ker ohranimo vse pretekle nize in jih lahko primerjamo z prihodnjimi nizi. A hkrati predstavlja slabost, ker zna slovar zelo hitro naraščati in zasesti celotni pomnilnik. Zato je v dejanskih implementacijah logično, da omejimo velikost slovarja. Prvotni opis algoritma LZ78 ne pojasnjuje, kaj narediti v primeru, ko zapolnimo celotni slovar, zato imamo več možnosti:

- Najpreprostejša rešitev je, da zamrznemo slovar. Od tega trenutka naprej ne dodajamo več novih vnosov v slovar in v bistvu postane statični slovar, vendar ga še zmeraj uporabljamo za kodiranje znakov.

- Izbrišemo celoten slovar in začnemo znova s praznim slovarjem. S to rešitvijo dejansko razdelimo izhodni tok na bloke, kjer ima vsak blok svoj lasten slovar. Ta način se uporablja najpogosteje.

- Iz slovarja zbrišemo nekaj najmanj uporabljenih vnosov, da pridobimo prostor za nove vnose. Ta način zna biti problematičen, ker ne vemo, koliko in katere nize izbrisati.

Kodirnik začne s slovarjem, v katerem vstavi znak `null` na ničto mesto. Med branjem vhodnega toka dodajamo nize v slovar od prvega mesta dalje po vrstnem redu branja. Npr., ko

preberemo znak `a` iz vhodnega toka, preiščemo celoten slovar, ali vsebuje enoznakovni niz `a`. Če ne najdemo nobenega vnosa, potem dodamo niz `a` na naslednje mesto v slovar in zapišemo žeton (`0`,`a`) v izhodni tok. Žeton z indeksom 0 zaznamuje, da znak `a` ni obstajal v slovarju. V nasprotnem primeru, če najdemo vnos v slovarju z nizom `a`, npr. na mestu 37, potem si zapomnimo ta indeks slovarja v neki spremenljivki I in nadaljujemo z branjem naslednjega znaka `b` iz vhodnega toka. Zdaj preiščemo slovar, ali vsebuje vnos z nizom I`b`, kjer I predstavlja niz iz slovarja, ki smo si ga zapomnili, torej v tem primeru je niz `ab`. Če ne najdemo vnosa, potem vstavimo I`b` oziroma niz `ab` na naslednje mesto slovarja in zapišemo žeton (`37`,`b`) v izhodni tok. Tak žeton predstavlja niz `ab`, ker se znak `a` nahaja v slovarju na mestu 37, ki ga združimo z znakom `b` iz žetona. Isti proces nadaljujemo, dokler ne preberemo vsega vhodnega toka.

**Primer 4.6**: Imamo slovar velikosti 16. Poskušamo zakodiramo niz `aacaacabcabacaddddc`.

| Korak | Indeks I | Vhodni znak x | Združen niz Ix | Niz Ix v slovarju? | Novi vnos v slovar | Izhodni žeton | Novi indeks |
|---|---|---|---|---|---|---|---|
| 1 | 0 | a | a | Ne | 1 – a | (0,a) | 0 |
| 2 | 0 | a | a | Da | / | / | 1 |
| 3 | 1 | c | ac | Ne | 2 – ac | (1,c) | 0 |
| 4 | 0 | a | a | Da | / | / | 1 |
| 5 | 1 | a | aa | Ne | 3 – aa | (1,a) | 0 |
| 6 | 0 | c | c | Ne | 4 – c | (0,c) | 0 |
| 7 | 0 | a | a | Da | / | / | 1 |
| 8 | 1 | b | ab | Ne | 5 – ab | (1,b) | 0 |
| 9 | 0 | c | c | Da | / | / | 4 |
| 10 | 4 | a | ca | Ne | 6 – ca | (4,a) | 0 |
| 11 | 0 | b | b | Ne | 7 – b | (0,b) | 0 |
| 12 | 0 | a | a | Da | / | / | 1 |
| 13 | 1 | c | ac | Da | / | / | 2 |
| 14 | 2 | a | aca | Ne | 8 – aca | (2,a) | 0 |
| 15 | 0 | d | d | Ne | 9 – d | (0,d) | 0 |
| 16 | 0 | d | d | Da | / | / | 9 |
| 17 | 9 | d | dd | Ne | 10 – dd | (9,d) | 0 |
| 18 | 0 | d | d | Da | / | / | 9 |
| 19 | 9 | c | dc | Ne | 11 – dc | (9,c) | 0 |

Tabela 4.6: Postopek kodiranja za Primer 4.6.

Iz Tabela 4.6 dobimo rezultat kodiranja, množico žetonov (`0`,`a`), (`1`,`c`), (`1`,`a`), (`0`,`c`), (`1`,`b`), (`4`,`a`), (`0`,`b`), (`2`,`a`), (`0`,`d`), (`9`,`d`), (`9`,`c`). Če upoštevamo, da so znaki veliki 8 bitov in indeksi 4 bite, izračunamo velikost zakodiranega niza: $11 \times (8 + 4) = 132$ bitov, kar je manj kot

originalna velikost 152 bitov. Razmerje stiskanja v tem primeru ni tako dobro kot pri prejšnjih dveh algoritmih, vendar je to zato, ker začnemo s praznim slovarjem in potrebujemo nekaj časa, da se napolni. Zato se algoritem LZ78 bolje pokaže, kadar imamo večji izvor za kodiranje.

**Primer 4.7**: Imamo slovar velikosti 16. Poskušamo dekodirati zakodiran niz (0,a), (1,c), (1,a), (0,c), (1,b), (4,a), (0,b), (2,a), (0,d), (9,d), (9,c) iz Primer 4.6.

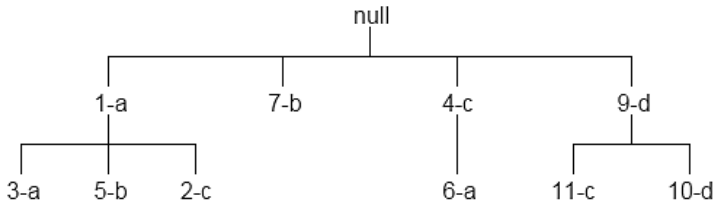
| Korak | Vhodni žeton | Niz iz slovarja | Novi vnos v slovar | Izhodni tok |
|---|---|---|---|---|
| 1 | (0,a) | / | 1 – a | a |
| 2 | (1,c) | a | 2 – ac | ac |
| 3 | (1,a) | a | 3 – aa | aa |
| 4 | (0,c) | / | 4 – c | c |
| 5 | (1,b) | a | 5 – ab | ab |
| 6 | (4,a) | c | 6 – ca | ca |
| 7 | (0,b) | / | 7 – b | b |
| 8 | (2,a) | aca | 8 – aca | aca |
| 9 | (0,d) | / | 9 – d | d |
| 10 | (9,d) | d | 10 – dd | dd |
| 11 | (9,c) | dc | 11 – dc | dc |

Tabela 4.7: Postopek dekodiranja za Primer 4.7.

Iz Tabela 4.7 dobimo rezultat dekodiranja, niz `aacaacabcabacaddddc`, ki se ujema s prvotnim nizom pred kodiranjem.

Dekodirnik mora graditi in vzdrževati svoj lasten slovar na isti način kot kodirnik, zato je precej bolj kompleksen za razliko od algoritma LZ77 in njegovih različic.

Za strukturo slovarja je priporočljivo izbrati številsko drevo. Drevo začnemo s korenom *null*. Vse znake, ki so v žetonu kazali na `null`, dodamo v drevo kot otroke korenu. Iz Primer 4.6 so to znaki `a`, `b`, `c` in `d`. In vsak od njih postane koren njihovim poddrevesom, kot vidimo na spodnji Slika 4.5. Nizi, ki se začnejo z znakom `a` (`aa`, `ab`, `ac` in `aca`), tvorijo poddrevo vozlišča `a`. Tako drevo nam omogoča enostaven način iskanja in dodajanje novih nizov.




Slika 4.5: Zgrajeno številsko drevo za Primer 4.6.

## 4.4 LZW

LZW je najpopularnejša različica algoritma LZ78, ki ga je razvil Terry A. Welch v članku [18] leta 1984. Glavna značilnost algoritma je odpravljanje drugega dela žetona, tako da žeton vsebuje le indeks slovarja. Zaradi tega moramo na začetku inicializirati slovar z vsemi znaki abecede. Pri tipičnih 8–bitnih znakih ASCII moramo v slovarju napolniti prvih 256 mest, ki ustrezajo vrstnemu redu kod ASCII, preden začnemo brati vhodni tok. Ker slovar inicializiramo z vsemi znaki abecede, bomo pri branju vhodnega toka že imeli posamezne znake v slovarju in zato potrebujemo v žetonu samo podatek o indeksu.

Kodirnik inicializira slovar in prazni niz I. Začnemo brati posamezne znake x iz vhodnega toka in preiščemo slovar, ali vsebuje niz Ix, kjer sta niz I in znak x združena. Če najdemo niz Ix v slovarju, potem nizu I dodelimo vrednost niza Ix in nadaljujemo z naslednjim vhodnim znakom. Kadar niza Ix ne najdemo v slovarju, naredimo sledeče: v izhodni tok zapišemo žeton z indeksom niza I iz slovarja, v slovar na naslednje mesto vstavimo vnos z nizom Ix in nizu I dodelimo vrednost znaka x.

**Primer 4.8**: Imamo slovar velikosti 512, ki ga inicializiramo od mesta 0 do 255 z znaki ASCII. Poskušamo zakodirati sporočilo `aacaacabcabacaddddc`.

| Korak | Niz I | Vhodni znak x | Združen niz Ix | Niz Ix v slovarju? | Novi vnos v slovar | Izhodni žeton | Novi niz I |
|---|---|---|---|---|---|---|---|
| 1 | / | a | a | Da | / | / | a |
| 2 | a | a | aa | Ne | 256 – aa | (97) | a |
| 3 | a | c | ac | Ne | 257 – ac | (97) | c |
| 4 | c | a | ca | Ne | 258 – ca | (99) | a |
| 5 | a | a | aa | Da | / | / | aa |
| 6 | aa | c | aac | Ne | 259 – aac | (256) | c |
| 7 | c | a | ca | Da | / | / | ca |
| 8 | ca | b | cab | Ne | 260 – cab | (258) | b |
| 9 | b | c | bc | Ne | 261 – bc | (98) | c |
| 10 | c | a | ca | Da | / | / | ca |
| 11 | ca | b | cab | Da | / | / | cab |
| 12 | cab | a | caba | Ne | 262 – caba | (260) | a |
| 13 | a | c | ac | Da | / | / | ac |
| 14 | ac | a | aca | Ne | 263 – aca | (257) | a |
| 15 | a | d | ad | Ne | 264 – ad | (97) | d |
| 16 | d | d | dd | Ne | 265 – dd | (100) | d |
| 17 | d | d | dd | Da | / | / | dd |
| 18 | dd | d | ddd | Ne | 266 – ddd | (265) | d |
| 19 | d | c | dc | Ne | 267 – dc | (100) | c |
| 20 | c | EOF | / | / | / | (99) | / |

Tabela 4.8: Postopek kodiranja za Primer 4.8.

Iz Tabela 4.8 dobimo rezultat kodiranja, množico žetonov (97), (97), (99), (256), (258), (98), (260), (257), (97), (100), (265), (100), (99). Konec vhodnega toka smo ponazorili z znakom EOF, kjer smo končali in zapisali zadnji žeton. Ker smo za naš primer izbrali 9–bitne žetone, končna velikost zakodiranega niza znaša $13 \times 9 = 117$ bitov v primerjavi s prvotno velikostjo 152 bitov. Praktične implementacije navadno izberejo žetone velikosti 16 bitov, kar znaša slovar z velikostjo 65536 vnosov.

Dekodirnik tudi začne z inicializacijo slovarja s prvimi 256 znaki abecede in praznima nizoma I in J. Začnemo z branjem žetona iz vhodnega toka, od katerega dobimo indeks, s katerim pridobimo niz iz slovarja in ga dodelimo nizu J ter zapišemo v izhodni tok. Nato združimo niz I in prvi znak niza J v novi niz $IJ_0$. Pri vsakem koraku dodamo niz $IJ_0$ v slovar na naslednje mesto, razen pri prvem koraku, saj zmeraj začnemo z nekim začetnim znakom, ki je že v slovarju. Na koncu moramo še nizu I dodeliti vrednost niza J. Celoten korak ponavljamo, dokler ne predelamo ves vhodni tok.

**Primer 4.9**: Imamo slovar velikosti 512, ki ga inicializiramo od mesta 0 do 255 z znaki ASCII. Poskušamo dekodirati žetone (97), (97), (99), (256), (258), (98), (260), (257), (97), (100), (265), (100) in (99) iz Primer 4.8.

| Korak | Niz I | Vhodni žeton | Niz J / Izhodni tok | Združen niz $IJ_0$ | Novi vnos v slovar | Novi niz I |
|---|---|---|---|---|---|---|
| 1 | / | (97) | a | / | / | a |
| 2 | a | (97) | a | aa | 256 – aa | a |
| 3 | a | (99) | c | ac | 257 – ac | c |
| 4 | c | (256) | aa | ca | 258 – ca | aa |
| 5 | aa | (258) | ca | aac | 259 – aac | ca |
| 6 | ca | (98) | b | cab | 260 – cab | b |
| 7 | b | (260) | cab | bc | 261 – bc | cab |
| 8 | cab | (257) | ac | caba | 262 – caba | ac |
| 9 | ac | (97) | a | aca | 263 – aca | a |
| 10 | a | (100) | d | ad | 264 – ad | d |
| 11 | d | (265) | dd | / | 265 – dd | dd |
| 12 | dd | (100) | d | ddd | 266 – ddd | d |
| 13 | d | (99) | c | dc | 267 – dc | c |

Tabela 4.9: Postopek dekodiranja za Primer 4.9.

Iz Tabela 4.9 dobimo končni rezultat dekodiranja: niz `aacaacabcabacaddddc`.

Pri dekodiranju lahko pride do velike izjeme, kar se je tudi nam zgodilo pri Tabela 4.9. Dobili smo žeton, ki kaže v slovarju na mesto 265, a tega vnosa slovar še nima. Do tega pride zato, ker dekodirnik zmeraj zaostaja za kodirnikom za en znak. Ta primer se vedno zgodi, kadar

kodirnik naleti na niz v obliki [znak][niz][znak][niz][znak], medtem ko je [znak][niz] že v slovarju, [znak][niz][znak] pa še ne. Nam se je to zgodilo zaradi kodiranja pri Tabela 4.8, kjer smo obdelovali niz `ddd`, a ravno pred tem smo v slovar dodali niz `dd`. Ta problem lahko dekodirnik reši tako, da uporabi zadnji uporabljen niz, ki mu doda še njegov prvi znak in ga uporabi za izhodni tok ter ga doda v slovar. To smo naredili tudi mi, tako da smo pri Tabela 4.9 dodelili nizu $J = II_0 =$ `dd`, zapisali `dd` v izhodni tok in ga vstavili v slovar.

Ker ima slovar omejeno velikost, tipično 65536 vnosov, se ta zelo hitro napolni, razen ob stiskanju zelo majhnega izvora. Torej imamo isti problem kot pri algoritmu LZ78, ki ga lahko rešimo ravno na enake načine, kot smo jih našteli v prejšnjem 4.3. Še ena zanimiva lastnost algoritma LZW je ta, da se nizi v slovarju povečajo samo za en znak, zato potrebuje dolgo časa, da dobimo dolge nize v slovarju in s tem možnost za doseganje dobrega razmerja stiskanja. Zato pravimo, da se LZW počasi prilagaja vhodnemu toku.

Algoritem LZW je bil velika tarča patentiranja v preteklosti. Leta 1985 je podjetje Sperry Corporation, v katerem je delal Terry Welch, patentiralo LZW. Naslednje leto je podjetje Unisys kupilo Sperry Corporation in s tem prevzelo lastništvo nad patentom. V letu 1987 je podjetje CompuServe zasnovalo format za slike GIF, kjer so za stiskanje uporabili algoritem LZW. Zdi se, da niso bili seznanjeni, da je bil algoritem LZW patentiran v tistem času in prav tako Unisys ni vedel, da so uporabili LZW v njihovem formatu GIF. Po letu 1987 je GIF pridobil na popularnosti in so ga mnogi razvijalci programske opreme začeli uporabljati v svojih programih. GIF je v tistem času tudi postal glavni format za prikazovanje slik na svetovnem spletu. Takrat je Unisys začel ukrepati in je zahteval od podjetja CompuServe, da plačajo licenco. CompuServe je leta 1994 poskušalo izpodbiti patent, a ameriško sodišče je priznalo patent podjetju Unisys, zato je podjetje CompuServe od takrat moralo plačevati licenco. Od takrat so vse komercialne programske opreme bile dolžne plačati licenco za uporabo algoritma LZW ali formata GIF. Za nekomercialne in neprofitne programske opreme ni bilo treba plačevati licenc. Leta 1995 so zato začeli razvijali novi brezizgubni format PNG za prikazovanje slik, ki ne vsebuje nobenih patentov. Še drugi znani primeri licenciranja algoritma LZW je bil tudi format TIFF, programski jezik PostScript in Unixov program compress za stiskanje podatkov. Leta 2003 in 2004 so vsi patenti potekli ([22], [26], [27]), tako da se zdaj lahko format GIF prosto uporablja brez kakršnih koli licenc [14].

# 5 Algoritem Burrows–Wheeler

Algoritem Burrows-Wheeler je bil predstavljen leta 1994 v tehničnem poročilu [6], ki sta ga napisala Michael Burrows in David J. Wheeler. Tehnično poročilo temelji na neobjavljenem delu Davida Wheelerja iz leta 1983. Na Slika 5.1 je prikazana struktura algoritma, kjer tudi vidimo, da je sestavljen iz več faz, ki se izvajajo zaporedoma:

- V prvi fazi naredimo transformacijo Burrows-Wheeler. Cilj te faze je preurediti vhodni tok tako, da so si identični znaki zelo blizu v končnem nizu.
- V drugi fazi uporabimo transformacijo premakni-na-začetek (angl. Move-To-Front, MTF). V tej fazi dodelimo znakom vhodnega toka indekse in ob procesiranju znake premikamo na začetek, zato da bolj pogosti znaki dobijo manjše indekse.
- V tretji fazi uporabimo katero koli entropijsko kodiranje za stiskanje.

Vhodni in izhodni tok vsake faze so bloki podatkov neke velikosti, ki jo določimo na začetku algoritma. V prvi in drugi fazi podatkov nič ne stisnemo in velikost ostane ista. Namen teh dveh transformacij je, da pripravimo podatke za kasnejše stiskanje z entropijskim kodirnikom za boljše razmerje stiskanja [7].


Slika 5.1: Struktura osnovnega algoritma Burrows-Wheeler.

## 5.1 Transformacija Burrows-Wheeler

Transformacija Burrows-Wheeler (angl. Burrows-Wheeler transform, BWT) nam omogoča, da naredimo permutacijo nad vhodnim nizom na tak način, da dobimo izhodni niz z razvrščenimi znaki, kjer so identični znaki grupirani oziroma so si zelo blizu. Presenetljiva lastnost transformacije je, da lahko iz permutacije povrnemo originalni niz, kljub temu da ima niz z n znaki n! permutacij. Tukaj so podatki zmeraj v obliki bloka podatkov, tako da je transformacija tudi znana kot bločno sortiranje (angl. block sorting).

Recimo, da je vhodni niz T velik n znakov, transformacijo začnemo tako, da ustvarimo matriko M velikosti $n \times n$. V prvo vrstico matrike M zapišemo vsebino niza T, kjer vsak znak stoji v svojem stolpcu in v vsaki novi vrstici naredimo krožno permutacijo prejšnje vrstice. Zatem razvrstimo vse vrstice v novo matriko M' glede na leksikografski vrstni red vrstic. V izhodni tok zapišemo znake iz zadnjega stolpca matrike M' skupaj s številko vrstice, ki vsebuje prvotni niz.

**Primer 5.1**: Imamo vhodni niz `banana`. Poskušamo izvesti transformacijo Burrows-Wheeler nad vhodnim nizom.

$$M = \begin{bmatrix} b & a & n & a & n & a \\ a & n & a & n & a & b \\ n & a & n & a & b & a \\ a & n & a & b & a & n \\ n & a & b & a & n & a \\ a & b & a & n & a & n \end{bmatrix} \xrightarrow{sort} M' = \begin{bmatrix} a & b & a & n & a & n \\ a & n & a & b & a & n \\ a & n & a & n & a & b \\ b & a & n & a & n & a \\ n & a & b & a & n & a \\ n & a & n & a & b & a \end{bmatrix} \tag{5.1}$$

Iz zadnjega stolpca matrike M' dobimo končni niz *C* = `nnbaaa` in indeks I = 3, ki predstavlja vrstico originalnega niza `banana`.

Če želimo iz niza C in indeksa I dobiti prvotni niz, moramo storiti obratno transformacijo. Ponovno ustvarimo prazno matriko M velikosti $n \times n$. Tokrat v zadnji stolpec zapišemo vsebino niza C, kjer so vsi znaki v lastni vrstici in razvrstimo matriko glede na leksikografski vrstni red vrstic. Nato vstavljamo niz C v zadnji prazen stolpec in leksikografsko razvrščamo, dokler ne zapolnimo in razvrstimo celotne matrike. Zatem v izhodni tok zapišemo znake iz vrstice, ki je enaka indeksu I.

**Primer 5.2**: Imamo vhodni niz C = `nnbaa` in indeks I = 3. Izvedemo obratno transformacijo Burrows-Wheeler.

$$\begin{array}{cccccc} \xrightarrow{add} & \begin{bmatrix} \\ \\ \\ \\ \\ \\ \end{bmatrix} & \begin{bmatrix} n \\ n \\ b \\ a \\ a \\ a \end{bmatrix} \xrightarrow{sort} & \begin{bmatrix} \\ \\ \\ \\ \\ \\ \end{bmatrix} & \begin{bmatrix} a \\ a \\ a \\ b \\ n \\ n \end{bmatrix} \xrightarrow{add} & \begin{bmatrix} n & a \\ n & a \\ b & a \\ a & b \\ a & n \\ a & n \end{bmatrix} \xrightarrow{sort} \\ \xrightarrow{sort} & \begin{bmatrix} \\ \\ \\ \\ \\ \\ \end{bmatrix} & \begin{bmatrix} a & b \\ a & n \\ a & n \\ b & a \\ n & a \\ n & a \end{bmatrix} \xrightarrow{add} & \begin{bmatrix} \\ \\ \\ \\ \\ \\ \end{bmatrix} & \begin{bmatrix} n & a & b \\ n & a & n \\ b & a & n \\ a & b & a \\ a & n & a \\ a & n & a \end{bmatrix} \xrightarrow{sort} & \begin{bmatrix} a & b & a \\ a & n & a \\ a & n & a \\ b & a & n \\ n & a & b \\ n & a & n \end{bmatrix} \xrightarrow{add} \end{array} \tag{5.2}$$

$$\xrightarrow{add}\begin{bmatrix} n & a & b & a \\ n & a & n & a \\ b & a & n & a \\ a & b & a & n \\ a & n & a & b \\ a & n & a & n \end{bmatrix}\xrightarrow{sort}\begin{bmatrix} a & b & a & n \\ a & n & a & b \\ a & n & a & n \\ b & a & n & a \\ n & a & b & a \\ n & a & n & a \end{bmatrix}\xrightarrow{add}\begin{bmatrix} n & a & b & a & n \\ n & a & n & a & b \\ b & a & n & a & n \\ a & b & a & n & a \\ a & n & a & b & a \\ a & n & a & n & a \end{bmatrix}\xrightarrow{sort}$$

$$\xrightarrow{sort}\begin{bmatrix} a & b & a & n & a \\ a & n & a & b & a \\ a & n & a & n & a \\ b & a & n & a & n \\ n & a & b & a & n \\ n & a & n & a & b \end{bmatrix}\xrightarrow{add}\begin{bmatrix} n & a & b & a & n & a \\ n & a & n & a & b & a \\ b & a & n & a & n & a \\ a & n & a & n & a & n \\ a & n & a & b & a & n \\ a & n & a & n & a & b \end{bmatrix}\xrightarrow{sort} M' = \begin{bmatrix} a & b & a & n & a & n \\ a & n & a & b & a & n \\ a & n & a & n & a & b \\ b & a & n & a & n & a \\ n & a & b & a & n & a \\ n & a & n & a & b & a \end{bmatrix}$$

Z zgornjim postopkom smo uspeli dobiti isto matriko M' kot v Primer 5.1. Zdaj s pomočjo indeksa I dobimo T = `banana` iz vrstice indeksa 3.

Zgornja transformacija temelji na osnovnem postopku, zapisanem v tehničnem poročilu. Transformacijo je možno tudi izboljšati na tak način, da ne potrebujemo indeksa I za rekonstrukcijo prvotnega niza. To storimo tako, da pri transformaciji dodamo nizu končni znak `$`, ki je manjše velikosti od vseh znakov abecede in ne obstaja v samem nizu. Znaka `$` dejansko ne izpišemo v izhodni tok pri razširjanju podatkov, temveč ga samo začasno uporabimo v stisnjenem toku.

**Primer 5.3**: Imamo vhodni niz `banana$`. Izvedemo transformacijo Burrows-Wheeler nad vhodnim nizom.

$$M = \begin{bmatrix} b & a & n & a & n & a & \$ \\ a & n & a & n & a & \$ & b \\ n & a & n & a & \$ & b & a \\ a & n & a & \$ & b & a & n \\ n & a & \$ & b & a & n & a \\ a & \$ & b & a & n & a & n \\ \$ & b & a & n & a & n & a \end{bmatrix}\xrightarrow{sort} M' = \begin{bmatrix} \$ & b & a & n & a & n & a \\ a & \$ & b & a & n & a & n \\ a & n & a & \$ & b & a & n \\ a & n & a & n & a & \$ & b \\ b & a & n & a & n & a & \$ \\ n & a & \$ & b & a & n & a \\ n & a & n & a & \$ & b & a \end{bmatrix} \tag{5.3}$$

Postopek transformacije in obratne transformacije ostane enak kot pri prejšnjih dveh primerih. V Primer 5.2 smo potrebovali indeks, da smo ugotovili, kje se nahaja naš originalni niz, vendar s pomočjo znaka `$` enostavno izvemo, da se nahaja v peti vrstici matrike M' pri izračunu (5.3), kajti niz se zmeraj konča z znakom `$`.

### 5.1.1 Transformacija s priponskim poljem

Izkaže se, da je transformacija Burrows-Wheeler povezana s priponskimi polji (angl. suffix array). Namreč s transformacijo nekega niza lahko dobimo njeno priponsko polje in obratno. Edini pogoj je, da uporabimo končni znak `$`. Na Slika 5.2 vidimo oba rezultata transformacije

M' in priponskega polja SA za niz `banana$` iz Primer 5.3. Priponsko polje dobimo tako, da v polje dodamo nize vrstic matrike, vendar samo do in vključno z znakom `$`.

$$M' = \begin{bmatrix} \$ & b & a & n & a & n & a \\ a & \$ & b & a & n & a & n \\ a & n & a & \$ & b & a & n \\ a & n & a & n & a & \$ & b \\ b & a & n & a & n & a & \$ \\ n & a & \$ & b & a & n & a \\ n & a & n & a & \$ & b & a \end{bmatrix} \qquad \text{SA} = \begin{array}{|c|l|} \hline 6 & \$ \\ \hline 5 & a\$ \\ \hline 3 & ana\$ \\ \hline 1 & anana\$ \\ \hline 0 & banana\$ \\ \hline 4 & na\$ \\ \hline 2 & nana\$ \\ \hline \end{array}$$

Slika 5.2: Primerjava med matriko in priponskem poljem za Primer 5.3.

Namesto prejšnjega postopka lahko torej transformacijo dobimo tudi tako, da zgradimo priponsko polje in iz njega razberemo končni niz C. Zgradimo priponsko polje SA iz vhodnega niza T, nize iz polja razvrstimo po leksikografskem vrstnem redu in se sprehodimo od začetnega niza naprej. Pri vsakem koraku i uporabimo indeks priponskega polja za določanje znaka na tem mestu v nizu C. Pomagamo si s pomočjo formule:

$$C[i] = \begin{cases} T[SA[i]-1], & SA[i] > 0 \\ \$, & SA[i] = 0 \end{cases} \tag{5.4}$$

**Primer 5.4**: Imamo začetni niz T = `banana$` in že zgrajeno priponsko polje iz Slika 5.2. Poskušamo dobiti transformacijo C od priponskega polja z enačbo (5.4).

| Korak | i | SA[i] | C[i] |
|---|---|---|---|
| 1 | 0 | 6 | a |
| 2 | 1 | 5 | n |
| 3 | 2 | 3 | n |
| 4 | 3 | 1 | b |
| 5 | 4 | 0 | $ |
| 6 | 5 | 4 | a |
| 7 | 6 | 2 | a |

Tabela 5.1: Postopek transformacije za Primer 5.4.

Iz Tabela 5.1 smo dobili transformacijo `annb$aa`, kar je enako kot pri matriki M' iz Slika 5.2. Ker smo pri Tabela 5.1 imeli SA[i] = 0, smo direktno zapisali znak `$`. S tem smo ponazorili premik znaka `$` na zadnjo mesto niza T.

### 5.1.2 Preslikava LF

Razlog, zakaj lahko povrnemo originalni niz iz permutacije, je zahvaljujoč pomembni lastnosti, ki jo imenujemo preslikava LF (angl. LF mapping – Last Front mapping). Ta

lastnost pravi, da imajo pojavitve znakov v zadnjem stolpcu enake range kakor pojavitve znakov prvega stolpca. Rang nekega znaka je številka, ki predstavlja, kolikokrat se je ta isti znak že pojavil v nizu. Npr. za niz `banana` bi predstavili range kot $b_0a_0n_0a_1n_1a_2$. Na Slika 5.3 je prikazana matrika M' iz izračuna (5.3), kjer so še dodatno določeni rangi vsem znakom glede na zadnji stolpec.

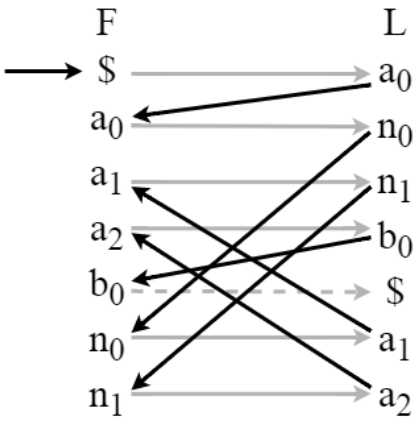
$$
M' = \begin{array}{ccccccc}
F & & & & & & L \\
\$ & b_0 & a_2 & n_1 & a_1 & n_0 & a_0 \\
a_0 & \$ & b_0 & a_2 & n_1 & a_1 & n_0 \\
a_1 & n_0 & a_0 & \$ & b_0 & a_2 & n_1 \\
a_2 & n_1 & a_1 & n_0 & a_0 & \$ & b_0 \\
b_0 & a_2 & n_1 & a_1 & n_0 & a_0 & \$ \\
n_0 & a_0 & \$ & b_0 & a_2 & n_1 & a_1 \\
n_1 & a_1 & n_0 & a_0 & \$ & b_0 & a_2
\end{array}
$$

Slika 5.3: Prikaz matrike M' z rangi znakov iz izračuna (5.3).

Znaku `$` ni treba podati ranga, saj se lahko pojavi le enkrat v nizu. Najpomembnejša sta prvi in zadnji stolpec, zato ostale notranje stolpce lahko spregledamo. Iz matrike vidimo, da imajo vsi isti znaki obeh stolpcev enak vrstni red rangov. Npr. vsi znaki `a` obeh stolpcev imajo naraščajoči vrstni red rangov 0, 1 in 2.

Vemo, da so znaki v prvem stolpcu razvrščeni po abecednem vrstnem redu, in ker imajo range v naraščajočem vrstnem redu, lahko pojavitve rangov prvega stolpca izračunamo v zadnjem stolpcu. To dejstvo nam omogoča, da opravimo obratno transformacijo precej hitreje v primerjavi s postopkom iz Primer 5.3. To lahko uporabimo le takrat, kadar smo v nizu dodali končni znak `$`. Postopek je tak, da zapišemo vhodni niz C v stolpec L skupaj z rangi znakov. Nato zapišemo niz C v stolpec F, ki ga razvrstimo po abecednem vrstnem redu prav tako z rangi znakov. Začnemo s prvim znakom stolpca F in ga vpišemo v prazni niz T. Nato pogledamo na istoležeči znak v stolpcu L in se premaknemo na ta znak v stolpcu F, ki ga dodamo in združimo z nizom T. Ponavljamo sprehajanje po stolpcih, dokler ne pridemo do znaka `$` v stolpcu L. Ostane nam samo še, da obrnemo vrstni red niza T vzvratno in s tem smo dobili originalni niz pred transformacijo.

**Primer 5.5**: Imamo vhodni niz C = `annb$aa`. Izvedemo obratno transformacijo nad vhodnim nizom s stolpcema LF.




Slika 5.4: Postopek obratne transformacije s stolpcema LF.

Iz Slika 5.4 dobimo rezultat sprehodov: niz T = `$ananab`. Nato moramo obrniti niz T vzvratno in s tem dobimo prvotni iskani niz T' = `banana$`.

## 5.2 Transformacija premakni-na-začetek

Namen transformacije premakni-na-začetek je, da vzdržuje seznam vseh znakov abecede, kjer se pogosto pojavljajoči znaki nahajajo proti začetku seznama. Algoritem so Jon Louis Bentley, Daniel D. Sleator, Robert E. Tarjan in Victor K. Wei objavili v članku [4] leta 1986. Transformacija ne stisne podatkov, temveč zmanjša redundanco, kadar imamo zaporedne ponovitve istih znakov. Zato ga uporabimo takoj po transformaciji Burrows-Wheeler in s tem izboljšamo razmerje stiskanja z entropijskim kodirnikom.

Na začetku transformacije ustvarimo razvrščen seznam vseh znakov abecede, v primeru znakov ASCII bi imeli seznam, velik 256 znakov, ki si sledijo naraščajoče glede na kode ASCII. Nato začnemo brati znake vhodnega toka, zapišemo indeks prebranega znaka v izhodni tok in premaknemo izpisan znak na začetek seznama. Postopek ponavljamo, dokler ne obdelamo vsega vhodnega toka. V bistvu gre za kodiranje znakov, kjer kode predstavljajo indeks seznama. Algoritem je lokalno adaptiven, saj se prilagaja glede na pogostost znakov iz vhodnega toka.

**Primer 5.6**: Imamo vhodni niz C = `cccaabdddaadcc` z abecedo znakov S = (`a`, `b`, `c`, `d`). Nad nizom izvedemo transformacijo premakni-na-začetek.

| Korak | Seznam | Vhodni znak | Izhodni znak | Posodobljeni seznam |
|---|---|---|---|---|
| 1 | abcd | c | 2 | cabd |
| 2 | cabd | c | 0 | cabd |
| 3 | cabd | c | 0 | cabd |
| 4 | cabd | a | 1 | acbd |
| 5 | acbd | a | 0 | acbd |
| 6 | acbd | b | 2 | bacd |
| 7 | bacd | d | 3 | dbac |
| 8 | dbac | d | 0 | dbac |
| 9 | dbac | d | 0 | dbac |
| 10 | dbac | a | 2 | adbc |
| 11 | adbc | a | 0 | adbc |
| 12 | adbc | d | 1 | dabc |
| 13 | dabc | c | 3 | cdab |
| 14 | cdab | c | 0 | cdab |

Tabela 5.2: Postopek transformacije premakni-na-začetek za Primer 5.6.

Iz Tabela 5.2 dobimo rezultat transformacije kode `2, 0, 0, 1, 0, 2, 3, 0, 0, 2, 0, 1, 3, 0`. Za naš primer smo uporabili abecedo velikosti 4, vendar bi v praksi uporabili velikosti 256 z vsemi kodami ASCII.

Postopek obratne transformacije je praktično enak kot pri transformaciji. Na začetku inicializiramo seznam vseh znakov abecede, ki so razvrščeni po svoji vrednosti. Preberemo kodo iz vhodnega toka, ki predstavlja indeks, nato v izhodni tok zapišemo znak iz seznama s tem indeksom in premaknemo izpisan znak na začetek seznama.

**Primer 5.7**: Imamo vhodne kode `2, 0, 0, 1, 0, 2, 3, 0, 0, 2, 0, 1, 3, 0` iz Primer 5.6 z abecedo znakov S = (`a, b, c, d`). Nad vhodnimi kodami izvedemo obratno transformacijo premakni-na-začetek.

| Korak | Seznam | Vhodna koda | Izhodni znak | Posodobljeni seznam |
|---|---|---|---|---|
| 1 | abcd | 2 | c | cabd |
| 2 | cabd | 0 | c | cabd |
| 3 | cabd | 0 | c | cabd |
| 4 | cabd | 1 | a | acbd |
| 5 | acbd | 0 | a | acbd |
| 6 | acbd | 2 | b | bacd |
| 7 | bacd | 3 | d | dbac |
| 8 | dbac | 0 | d | dbac |
| 9 | dbac | 0 | d | dbac |
| 10 | dbac | 2 | a | adbc |
| 11 | adbc | 0 | a | adbc |
| 12 | adbc | 1 | d | dabc |
| 13 | dabc | 3 | c | cdab |
| 14 | cdab | 0 | c | cdab |

Tabela 5.3: Postopek obratne transformacije premakni-na-začetek za Primer 5.7.

Iz Tabela 5.3 dobimo rezultat obratne transformacije: niz `cccaabdddaadcc`, kar je tudi naš originalni niz.

Do zdaj smo opisali originalno implementacijo transformacije, a obstajajo tudi številne različice algoritma, ki se razlikujejo v načinu posodabljanja seznama. Glavni razlog za uporabo transformacije je dobiti dolge ponovitve istih kod `0`, in če se med ponovitvami istih znakov znajde le en drugačen znak, ta precej zmanjša dolžino ponovitve in hkrati učinkovitost stiskanja. Zato lahko uporabimo malo drugačno transformacijo, kajti premakni-na-začetek (angl. Move-To-Front) je ena izmed več hevrističnih načinov za samoorganiziranje seznamov (angl. Self-Organizing List). Drugi znani načini so:

- Prenos (angl. Transpose): znak zamenjamo s prejšnjim znakom, če obstaja tak znak.
- Premakni-naprej-k (angl. Move-Ahead-k): znak premaknemo za k mest nazaj. Če spremenljivki k dodelimo kar indeks znaka, potem je enako kot način Move-To-Front, ali če spremenljivki k dodelimo 1, potem je enako kot način Transpose.
- Počakaj-c-in-premakni (angl. Wait-c-And-Move): znak premaknemo na začetek seznama, takrat ko se ta znak pojavi vsaj c-krat. Tukaj moramo dodatno spremljati, kolikokrat so se posamezni znaki pojavili.
- Štetje (angl. Count): znake seznama razvrščamo glede na število pojavitev v padajočem vrstnem redu. Tudi tukaj moramo dodatno spremljati, kolikokrat so se posamezni znaki pojavili.

## 5.3 Entropijsko kodiranje

Za dejansko stiskanje algoritma Burrow-Wheeler uporabimo entropijski kodirnik. Najbolj uporabljena sta Huffmanovo in aritmetično kodiranje. Na začetku Algoritem Burrows–Wheeler smo navedli originalen algoritem, kakršnega sta zasnovala Burrows in Wheeler. V njunem poročilu sta za entropijsko kodiranje uporabila modificirano Huffmanovo kodiranje. Omenila sta, da bi uporaba kodirnika RLE pred entropijskem kodiranjem lahko pripomogla k učinkovitosti stiskanja zaradi dolgih ponovitev kod `0` po transformaciji premakni-na-začetek. Mnoge implementacije različic Burrows-Wheelerja tudi uporabijo kodiranje RLE kot dodaten korak, poleg drugih izboljšav. Program Bzip2, ki implementira algoritem Burrows-Wheeler, uporabi kodiranje RLE dvakrat, in sicer pred transformacijo Burrows-Wheeler in po transformaciji premakni-na-začetek.

# 6 Implementacija in testiranje algoritmov v sistemu ALGator

## 6.1 Implementacija algoritmov

Za testiranje smo uporabili že implementirane odprtokodne algoritme za stiskanje podatkov v programskem jeziku Java. Testirali smo implementacije tistih algoritmov, ki smo jih opisali v prejšnjih poglavjih in hkrati ki se uporabljajo v današnjem času. Te implementacije so:

- Huffmanovo kodiranje – tukaj je algoritem implementiran z adaptivnim stiskanjem, kar pomeni, da sestavlja verjetnostno drevo dinamično in zato ne rabi brati vhodnega toka dvakrat.
- Aritmetično kodiranje – algoritem je implementiran z adaptivnim modelom unigram. Tudi tukaj algoritem prebere vhodni tok le enkrat.
- LZSS – implementacija uporablja krožno vrsto za iskalni medpomnilnik. Žetoni so veliki 16 bitov, kjer so 12-bitni uporabljeni za iskalni medpomnilnik in 4-bitni za vpogledni medpomnilnik.
- LZW – algoritem je implementiran z uporabo žetonov spremenljive dolžine. Za testiranje smo uporabili začetno velikost 9 bitov in največjo velikost 21 bitov. Ko se medpomnilnik napolni, neha dodajati nove vnose v slovar.
- Burrows-Wheeler – algoritem uporablja implementacijo programa Bzip2, ki poleg transformacij Burrows-Wheeler in premakni-na-začetek ter Huffmanovega kodiranja uporablja tudi kodiranje RLE.

Vse implementacije uporabljajo tokove za branje in pisanje podatkov, torej razred `InputStream` za vhodni tok in `OutputStream` za izhodni tok. Ker so algoritmi implementirani na različne načine, smo vsem dodali enotni razred, prek katerega sistem ALGator kliče metodi za stiskanje in razširjanje, kjer smo za vhodni in izhodni tok uporabili razširjena razreda `BufferedInputStream` in `BufferedOutputStream` za boljšo učinkovitost. Velikost obeh medpomnilniških tokov smo določili 8192 bajtov. Izvorna koda za celotni projekt je dostopna na spletni strani GitHuba [28].

## 6.2 Sistem ALGator

Testiranje je bilo izvedeno s pomočjo sistema ALGator [29], ki je namenjen za izvajanje algoritmov na testnih podatkih in analizo rezultatov izvajanja. Sistem nam omogoča ustvarjanje projekta, v katerem definiramo problem, določimo testne zbirke in izvajamo delovanje algoritmov nad testnimi zbirkami. V enem projektu je možno ustvariti poljubno število algoritmov za reševanje danega problema. Po izvajanju algoritmov lahko s sistemom analiziramo rezultate in primerjamo učinkovitost posameznih algoritmov.

ALGator je možno uporabljati na dva načina: samostojno ali kot spletno aplikacijo. Za naše potrebe je bilo zadosti uporaba kot samostojne aplikacije. Sistem je tudi neodvisen od platforme in je prenosljiv, saj se nobena izmed njegovih komponent ne namešča na sami operacijski sistem.

## 6.3 Način testiranja

Testiranje je bilo izvedeno na osebnem računalniku z naslednjimi specifikacijami:

- štirijedrni procesor Intel Core i5 760 2.80 GHz,
- pomnilnik 4 GB 1333 MHz,
- operacijski sistem Ubuntu 15.10 "Wily Werewolf" Minimal 64-bit,
- razvojno okolje Oracle Java SE JDK 8u72.

Izbrali smo minimalno inštalacijo operacijskega sistema Linux brez namiznega okolja z namenom, da maksimiziramo delovanje računalnika samo za namen testiranja. To smo dosegli tako, da smo spremenili procesorsko razporejanje, kjer smo procesu ALGatorja podali najvišjo procesorsko prioriteto -20 z Linuxovim ukazom `nice`.

Hoteli smo tudi odpraviti vpliv delovanja trdega diska, zato smo za lokacijo stiskanja in razširjanja uporabili začasni datotečni sistem tmpfs, ki se v celoti nahaja v pomnilniku. Ker se datoteke zapisujejo v pomnilnik, smo morali izklopiti Swap particijo, da se ne bi pomnilniške strani zamenjale med samim zapisovanjem.

ALGator prav tako omogoča večkratno izvajanje istih testov in zato ponuja izbiro več časov izvajanja, kot so najmanjši, največji, skupni in povprečni čas. Za naše testiranje smo vse testne zbirke izvajali kar trikrat in za rezultate izbrali najmanjše vrednosti časa stiskanja in razširjanja.

## 6.4 Testne zbirke

Tukaj so naštete vse testne zbirke, ki smo jih uporabili za ocenjevanje učinkovitosti algoritmov. Vsaka zbirka je zasnovana tako, da vsebuje več vrst tipičnih datotek, ki se jih pogosto uporablja pri stiskanju. Pomembna lastnost testnih zbirk je tudi, da se vsebina zbirk ne spreminja in zmeraj ostanejo identične.

### *6.4.1 Calgary Corpus*

Zbirko so ustvarili Ian Witten, Timothy Bell in John Cleary leta 1987 z namenom uporabiti v njihovem raziskovalnem članku [2] leta 1989 in knjigi [3] leta 1990. Po tem letu je Calagry Corpus postala standardna zbirka, s katero so ostali raziskovalci testirali in primerjali algoritme za brezizgubno stiskanje.

| Datoteka | Velikost (B) | Opis |
|---|---|---|
| bib | 111.261 | Bibliografski podatki v formatu refer |
| book1 | 768.771 | Knjiga "Far from the Madding Crowd" |
| book2 | 610.856 | Knjiga "Principles of Computer Speech" v formatu troff |
| geo | 102.400 | Seizmični podatki v obliki 32–bitnih števil s plavajočo vejico |
| news | 377.109 | Tekstovne novice v formatu Usenet |
| obj1 | 21.504 | Izvršljiv program datoteke progp z nabori ukazov VAX |
| obj2 | 246.814 | Izvršljiv program za Apple Macintosh |
| paper1 | 53.161 | Članek "Arithmetic coding for data compression" v formatu troff |
| paper2 | 82.199 | Članek "Computer (In)security: Infiltrating Open Systems" v formatu troff |
| paper3 | 46.526 | Članek "In Search of Autonomy" v formatu troff |
| paper4 | 13.286 | Članek "Programming by Example Revisited" v formatu troff |
| paper5 | 11.954 | Članek "A Logical Implemntation of Arithmetic" v formatu troff |
| paper6 | 38.105 | Članek "Compact Hash Tables Using Bidirectional Linear Probing" v formatu troff |
| pic | 513.216 | Črno-bela bitna slika v formatu CCITT faksimile |
| progc | 39.611 | Izvorna koda v jeziku C |
| progl | 71.646 | Izvorna koda v jeziku Lisp |
| progp | 49.379 | Izvorna koda v jeziku Pascal |
| trans | 93.695 | Transkript terminalske seje |

Tabela 6.1: Zbirka Calgary Corpus.

Velika zbirka Calgary Corpus vsebuje vseh 18 datotek, medtem ko standardna zbirka Calgary Corpus vsebuje 14 datotek, namreč brez datotek paper3, paper4, paper5, paper6. Te datoteke niso priložene, ker se je izkazalo, da nič ne doprinesejo k ocenjevanju algoritmov. V Tabela 6.1 je prikazana velika zbirka korpusa. Za naše testiranje smo uporabili standardno zbirko s 14 datotekami.

### 6.4.2 Canterbury Corpus

Zbirko sta objavila Ross Arnold in Timothy Bell v članku [1] leta 1997 kot izboljšano različico prejšnje zbirke Calgary Corpus. Od nabora več kot 800 datotek, primernih za vključitev v zbirko, sta izbrala 11 končnih datotek, ki so opisane v Tabela 6.2. V članku je razloženo, kako sta izbrala te datoteke in zakaj je težko najti datoteke, primerne za testiranje.

| Datoteka | Velikost (B) | Opis |
|---|---|---|
| alice29.txt | 152.089 | Knjiga "Alice's Adventures in Wonderland" |
| asyoulik.txt | 125.179 | Igra "As You Like It" |
| cp.html | 24.603 | Datoteka HTML |
| fields.c | 11.150 | Izvorna koda v jeziku C |
| grammar.lsp | 3.721 | Izvorna koda v jeziku Lisp |
| kennedy.xls | 1.029.744 | Excel datoteka |
| lcet10.txt | 426.754 | Članek "Workshop on Electronic Texts Proceedings" |
| plrabn12.txt | 481.861 | Pesem "Paradise Lost" |
| ptt5 | 513.216 | Črno-bela bitna slika v formatu CCITT faksimile |
| sum | 38.240 | Izvršljiv program v zbirnem jeziku SPARC |
| xargs.1 | 4.227 | GNU manual page v formatu troff |

Tabela 6.2: Zbirka Canterbury Corpus.

### 6.4.3 Silesia Corpus

Zbirko je ustvaril Sebastian Deorowicz [33] leta 2003. Namen zbirke je zagotoviti množico datotek, ki predstavljajo bolj sodobne tipe podatkov. Poudaril je pomanjkljivosti prejšnjih testnih zbirk in izboljšal zbirko Silesia Corpus z uporabo nasploh večjih velikosti datotek, dodatnega neangleškega besedila in podatkovnih baz. V Tabela 6.3 so opisane vse datoteke.

| Datoteka | Velikost (B) | Opis |
|---|---|---|
| dickens | 10.192.446 | Zbirka knjig Charlesa Dickensa iz projekta Gutenberg |
| mozilla | 51.220.480 | Izvršljiva inštalacija brskalnika Mozzila 1.0 v arhivski datoteki Tar |
| mr | 9.970.564 | Medicinska slika magnetne resonance v formatu DICOM |
| nci | 33.553.445 | Podatkovna baza kemijskih struktur v formatu SDF |
| ooffice | 6.152.192 | Dinamična knjižnica za program OpenOffice 1.01 |
| osdb | 10.085.684 | Podatkovna baza v formatu MySQL |
| reymont | 6.627.202 | Knjiga Chłopi v formatu PDF |
| samba | 21.606.400 | Izvorna koda projekta Samba 2.2-3 v arhivski datoteki Tar |
| sao | 7.251.944 | Katalog zvezd SAO |
| webster | 41.458.703 | Websterjev angleški slovar iz leta 1913 iz projekta Gutenberg v formatu HTML |
| xml | 5.345.280 | Zbirka datotek XML |
| x–ray | 8.474.240 | Rentgenska medicinska slika |

Tabela 6.3: Zbirka Silesia Corpus.

### 6.4.4 Maximum Compression

Zbirko je ustvaril Werner Bergmans [31] leta 2003 za testiranje različnih programov za brezizgubno stiskanje. Vse do leta 2011 je opravil testiranje več kot 150 programov. Zbirka vsebuje datoteke z različnimi tipi podatkov, katerih opis vidimo v Tabela 6.4.

| Datoteka | Velikost (B) | Opis |
|---|---|---|
| A10.jpg | 842.468 | Bitna slika v formatu JPEG |
| AcroRd32.exe | 3.870.784 | Izvršljiv program Acrobat Reader 5.0 |
| english.dic | 4.067.439 | Razvrščen seznam angleških besed (word list) |
| FlashMX.pdf | 4.526.946 | Datoteka PDF |
| FP.LOG | 20.617.071 | Prometni dnevnik za spletno stran Fighter-planes.com |
| MSO97.DLL | 3.782.416 | Dinamična knjižnica za program Microsoft Office 97 |
| ohs.doc | 4.168.192 | Dokument v formatu Microsoft Word |
| rafale.bmp | 4.149.414 | Bitna slika |
| vcfiu.hlp | 4.121.418 | Opisna datoteka za program Delphi First Impression v formatu WinHelp |
| world95.txt | 2.988.578 | Svetovni referenčni vir držav za leto 1995 iz projekta Gutenberg |

Tabela 6.4: Zbirka Maximum Compression.

## 6.5 Rezultati testiranja

Za preverjanje učinkovitosti izvajanja algoritmov smo s sistemom ALGator izmerili čas stiskanja, čas razširjanja in razmerje stiskanja. Časa sta izmerjena v milisekundah (ms), medtem ko je razmerje predstavljeno v odstotkih, in sicer kot:

$$razmerje\ stiskanja = \frac{velikost\ stisnjene\ datoteke}{velikost\ prvotne\ datoteke} \times 100 \qquad (6.1)$$

Manjše razmerje kot dobimo, učinkovitejši je algoritem. Ker bomo primerjali algoritme med seboj na več različnih datotekah hkrati, ki so različnih velikosti, smo hitrost stiskanja in razširjanja izračunali kot:

$$hitrost\ stiskanja/razširjanja\ = \frac{čas\ stiskanja/razširjanja}{velikost\ prvotne\ datoteke} \qquad (6.2)$$

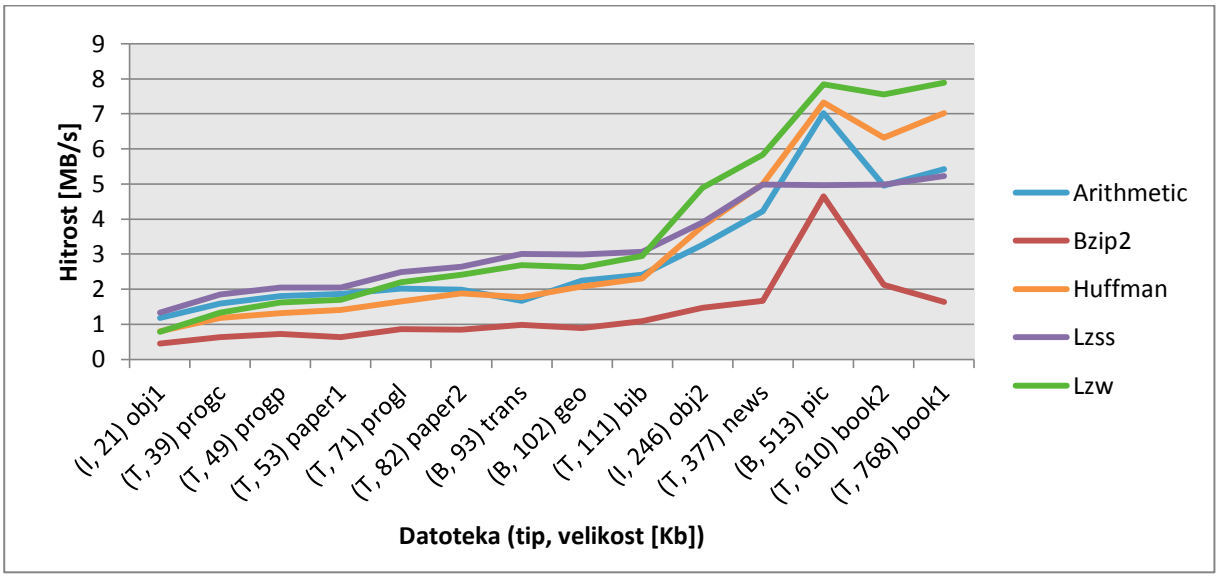
Hitrost merimo v MB/s, kar nam pove, koliko podatkov MB algoritem obdela v eni sekundi. Tako merilo nam omogoča primerjati čase algoritmov na datotekah z različnimi velikostmi. Ker so datoteke tudi različnih formatov, smo jim določili okrajšave glede na njihov tip podatkov: (T)ekstovna datoteka, (B)inarna datoteka, (I)zvršljiva datoteka in (Z)družena datoteka (Tar). Imenom datotek smo poleg tipa podali tudi njihovo velikost v Kb.

### 6.5.1 Analiza rezultatov na zbirki Calgary

V Tabela 6.5 je izpisan čas stiskanja za vseh pet algoritmov. Ker so datoteke različnih velikosti, smo podatke v vseh tabelah in grafih razvrstili po velikosti datotek naraščajoče. Časi z najmanjšo vrednostjo so označeni z odebeljeno pisavo, kjer tudi opazimo, da ima algoritem Lzss najhitrejši čas stiskanja za prvih 9 datotek, algoritem Lzw pa za ostale datoteke. Na Slika 6.1 je narisan graf, ki prikazuje primerjavo časov stiskanja. Iz grafa razberemo, da je Bzip2 najpočasnejši pri vseh datotekah. Vidimo tudi, da postajajo z večanjem velikosti datotek vsi algoritmi učinkovitejši. Edino neskladnost opazimo pri Lzss za datoteko pic.

| | Čas stiskanja [ms] | | | | |
|---|---|---|---|---|---|
| Datoteka | Arithmetic | Bzip2 | Huffman | Lzss | Lzw |
| obj1 | 18,3 | 47,2 | 26,7 | **16,0** | 27,1 |
| progc | 24,7 | 63,0 | 33,4 | **21,4** | 29,5 |
| progp | 27,3 | 67,6 | 37,6 | **24,0** | 30,2 |
| paper1 | 28,5 | 83,3 | 37,8 | **25,9** | 31,2 |
| progl | 35,5 | 83,0 | 43,4 | **28,7** | 32,4 |
| paper2 | 41,4 | 97,1 | 43,7 | **31,0** | 34,0 |
| trans | 56,1 | 95,7 | 52,8 | **31,1** | 34,8 |
| geo | 45,4 | 115,6 | 49,3 | **34,1** | 39,0 |
| bib | 45,9 | 101,4 | 48,2 | **36,3** | 37,7 |
| obj2 | 75,6 | 167,5 | 64,9 | 63,1 | **50,4** |
| news | 89,2 | 226,1 | 75,4 | 75,5 | **64,6** |
| pic | 73,0 | 110,5 | 70,0 | 103,3 | **65,3** |
| book2 | 123,2 | 288,2 | 96,5 | 122,6 | **80,8** |
| book1 | 141,7 | 469,7 | 109,3 | 147,1 | **97,4** |

Tabela 6.5: Rezultati časov stiskanja za zbirko Calgary.

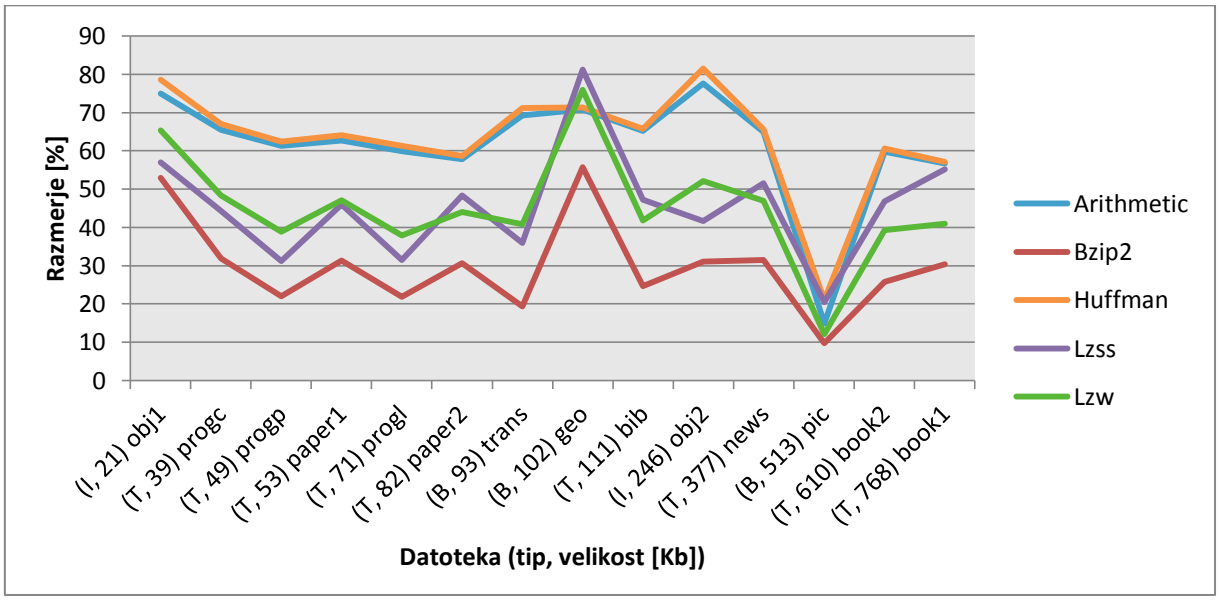



Slika 6.1: Grafični prikaz primerjave hitrosti stiskanja za zbirko Calgary.

Zdaj, ko poznamo hitrosti stiskanja, lahko primerjamo algoritme, kako dobro so stisnili datoteke. Iz Tabela 6.6 je razvidno, da Bzip2 proizvede stisnjene datoteke z najboljšim razmerjem za vse datoteke. Slika 6.2 nam pove tudi, da so razmerja precej sorazmerna med Bzip2, Lzss in Lzw. Enako velja za aritmetično in Huffmanovo kodiranje, kjer sta razmerji med tema dvema skoraj enaka. Vsi algoritmi so pridobili največjo kompresijo pri datoteki pic, ta datoteka je namreč črno-bela slika in je zato skoraj v celoti (87 %) sestavljena iz bajtov dolžine 0. Zanimiva je datoteka geo, kjer so Bzip2, Lzss in Lzw dobili precej slabo razmerje.

| | Razmerje [%] | | | | |
|---|---|---|---|---|---|
| Datoteka | Arithmetic | Bzip2 | Huffman | Lzss | Lzw |
| obj1 | 74,96 | **52,96** | 78,59 | 56,95 | 65,33 |
| progc | 65,56 | **31,93** | 67,07 | 44,26 | 48,33 |
| progp | 61,34 | **22,08** | 62,45 | 31,28 | 38,9 |
| paper1 | 62,74 | **31,38** | 64,17 | 46,02 | 47,17 |
| progl | 59,97 | **21,94** | 61,37 | 31,43 | 37,89 |
| paper2 | 57,8 | **30,67** | 58,65 | 48,3 | 43,99 |
| trans | 69,28 | **19,39** | 71,22 | 35,9 | 40,81 |
| geo | 70,74 | **55,76** | 71,39 | 81,23 | 75,96 |
| bib | 65,26 | **24,63** | 65,75 | 47,27 | 41,82 |
| obj2 | 77,66 | **31,1** | 81,53 | 41,73 | 52,12 |
| news | 64,88 | **31,5** | 65,68 | 51,56 | 46,97 |
| pic | 14,94 | **9,79** | 21,1 | 20,52 | 12,12 |
| book2 | 59,79 | **25,8** | 60,61 | 46,81 | 39,35 |
| book1 | 56,66 | **30,35** | 57,12 | 55,17 | 40,92 |

Tabela 6.6: Rezultati razmerij stiskanja za zbirko Calgary.

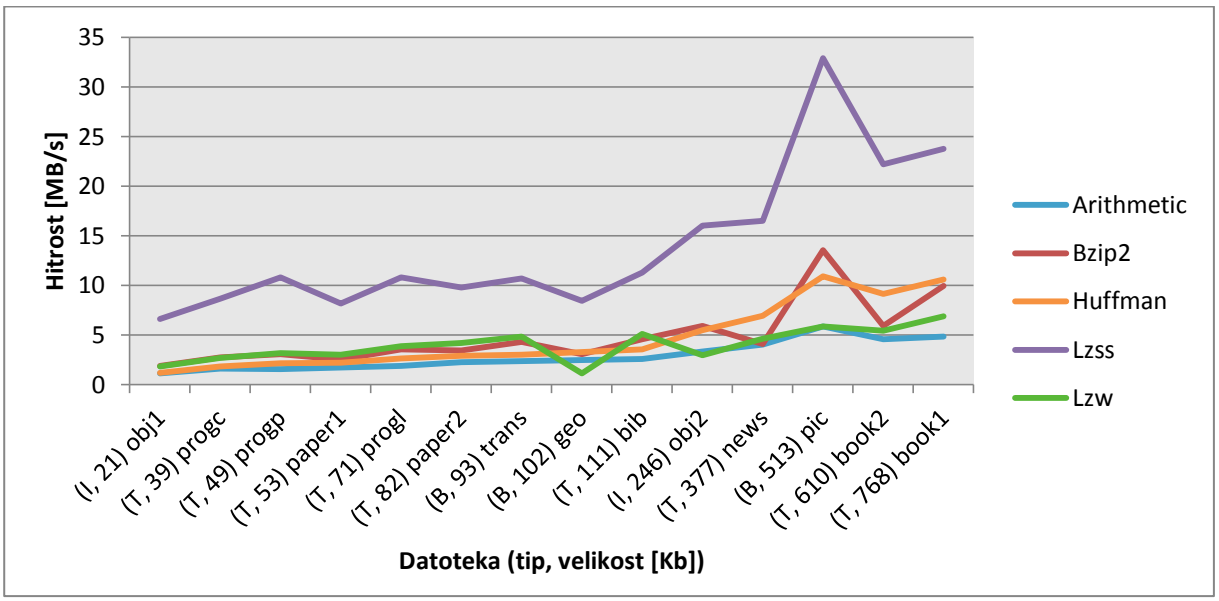



Slika 6.2: Grafični prikaz primerjave razmerij stiskanja za zbirko Calgary.

Kot vidimo v Tabela 6.7 in še posebno na Slika 6.3, pri razširjanju je nedvomno najhitrejši algoritem Lzss. Tudi tukaj vsi algoritmi postanejo izredno hitri pri datoteki pic.

| | Čas razširjanja [ms] | | | | |
|---|---|---|---|---|---|
| Datoteka | Arithmetic | Bzip2 | Huffman | Lzss | Lzw |
| obj1 | 18,9 | 11,5 | 17,8 | **3,2** | 11,8 |
| progc | 24,3 | 14,2 | 21,6 | **4,5** | 14,7 |
| progp | 31,5 | 16,0 | 23,0 | **4,5** | 15,5 |
| paper1 | 30,6 | 20,7 | 24,0 | **6,4** | 17,6 |
| progl | 37,4 | 20,1 | 26,8 | **6,6** | 18,4 |
| paper2 | 35,9 | 23,6 | 28,2 | **8,3** | 19,4 |
| trans | 39,3 | 21,6 | 30,9 | **8,7** | 19,3 |
| geo | 41,3 | 33,2 | 30,9 | **12,0** | 88,4 |
| bib | 42,8 | 24,2 | 31,3 | **9,8** | 21,7 |
| obj2 | 73,6 | 41,8 | 44,7 | **15,3** | 83,5 |
| news | 92,7 | 91,4 | 54,4 | **22,8** | 81,7 |
| pic | 87,1 | 37,8 | 46,9 | **15,5** | 87,0 |
| book2 | 132,6 | 103,3 | 66,9 | **27,4** | 112,6 |
| book1 | 158,8 | 77,0 | 72,4 | **32,3** | 111,5 |

Tabela 6.7: Rezultati časov razširjanja za zbirko Calgary.



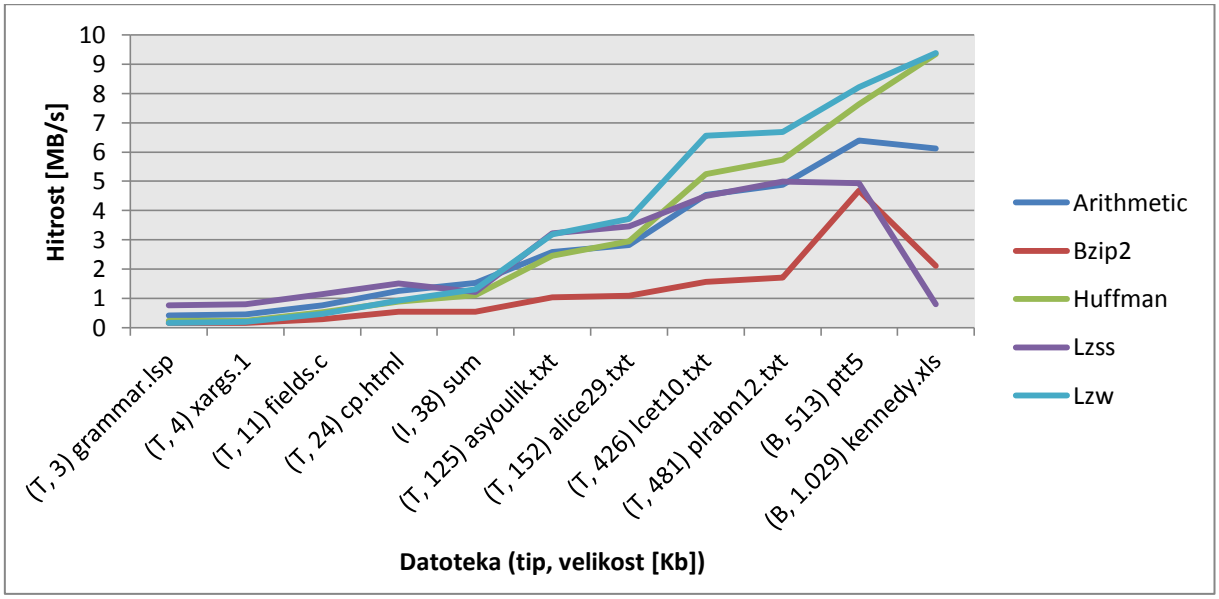

Slika 6.3: Grafični prikaz primerjave hitrosti razširjanja za zbirko Calgary.

### 6.5.2 Analiza rezultatov na zbirki Canterbury

Podobno kot pri prejšnji zbirki Calgary iz Tabela 6.8 vidimo, da je Lzss najhitrejši pri stiskanju datotek manjših velikosti, medtem ko je Lzw hitrejši pri večjih datotekah. Izjema je datoteka sum, ki jo je aritmetično kodiranje stisnilo najhitreje. Tudi tokratni graf na Slika 6.4 prikazuje naraščanje hitrosti stiskanja pri večanju velikosti datotek, razen za datoteko kennedy.xls, kjer se algoritma Bzip2 in Lzss ne pokažeta najbolje. Tukaj je Bzip2 tudi najpočasnejši za kar velik odstotek pri srednje velikih datotekah.

| | Čas stiskanja [ms] | | | | |
|---|---|---|---|---|---|
| Datoteka | Arithmetic | Bzip2 | Huffman | Lzss | Lzw |
| grammar.lsp | 9,1 | 23,4 | 16,7 | **4,9** | 22,8 |
| xargs.1 | 9,5 | 25,6 | 18,2 | **5,2** | 23,0 |
| fields.c | 14,5 | 38,6 | 21,7 | **9,8** | 24,3 |
| cp.html | 19,6 | 45,9 | 27,5 | **16,2** | 26,5 |
| sum | **25,0** | 70,4 | 34,3 | 31,5 | 29,2 |
| asyoulik.txt | 48,4 | 121,0 | 51,0 | **38,9** | 39,2 |
| alice29.txt | 53,8 | 139,4 | 51,5 | 44,0 | **41,0** |
| lcet10.txt | 94,2 | 274,3 | 81,3 | 95,0 | **65,1** |
| plrabn12.txt | 98,8 | 281,2 | 83,9 | 96,6 | **72,0** |
| ptt5 | 80,2 | 109,8 | 67,2 | 104,1 | **62,5** |
| kennedy.xls | 168,3 | 487,2 | 110,1 | 1.285,7 | **109,6** |

Tabela 6.8: Rezultati časov stiskanja za zbirko Canterbury.

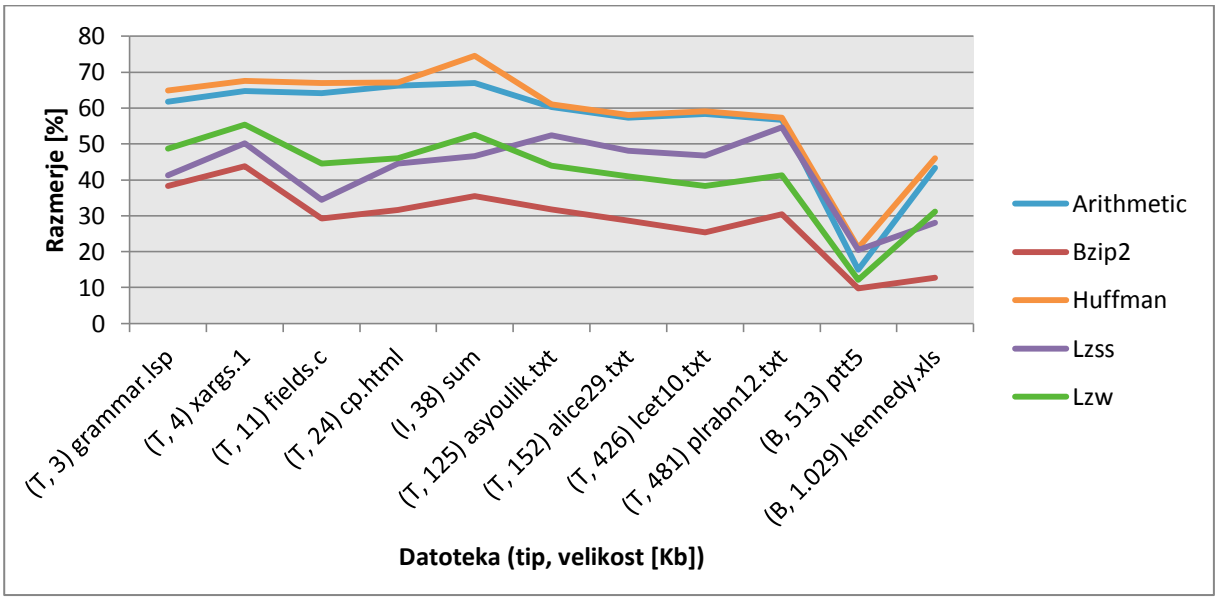



Slika 6.4: Grafični prikaz primerjave hitrosti stiskanja za zbirko Canterbury.

Iz Tabela 6.9 ugotovimo, da ima Bzip2 ponovno najboljša razmerja stiskanja. Na Slika 6.5 vidimo pri datoteki ptt5 precejšen skok v učinkovitosti, ki je pravzaprav ista črno-bela slika kot datoteka pic v prejšnji zbirki Calgary, tako da ne preseneča, da je tudi tukaj dosegla zelo dobro razmerje. Najslabša razmerja je zopet doseglo Huffmanovo kodiranje.

| | Razmerje [%] | | | | |
|---|---|---|---|---|---|
| Datoteka | Arithmetic | Bzip2 | Huffman | Lzss | Lzw |
| grammar.lsp | 61,78 | **38,24** | 64,82 | 41,31 | 48,7 |
| xargs.1 | 64,75 | **43,86** | 67,57 | 50,25 | 55,33 |
| fields.c | 64,2 | **29,25** | 66,95 | 34,45 | 44,52 |
| cp.html | 66,22 | **31,61** | 67,18 | 44,47 | 46 |
| sum | 67,03 | **35,43** | 74,61 | 46,56 | 52,57 |
| asyoulik.txt | 60,34 | **31,8** | 61,02 | 52,37 | 43,93 |
| alice29.txt | 57,29 | **28,61** | 58,06 | 48,08 | 40,93 |
| lcet10.txt | 58,31 | **25,3** | 59,04 | 46,8 | 38,24 |
| plrabn12.txt | 56,74 | **30,4** | 57,34 | 54,57 | 41,19 |
| ptt5 | 14,94 | **9,79** | 21,1 | 20,52 | 12,12 |
| kennedy.xls | 43,31 | **12,74** | 46,04 | 27,98 | 31,15 |

Tabela 6.9: Rezultati razmerij stiskanja za zbirko Canterbury.

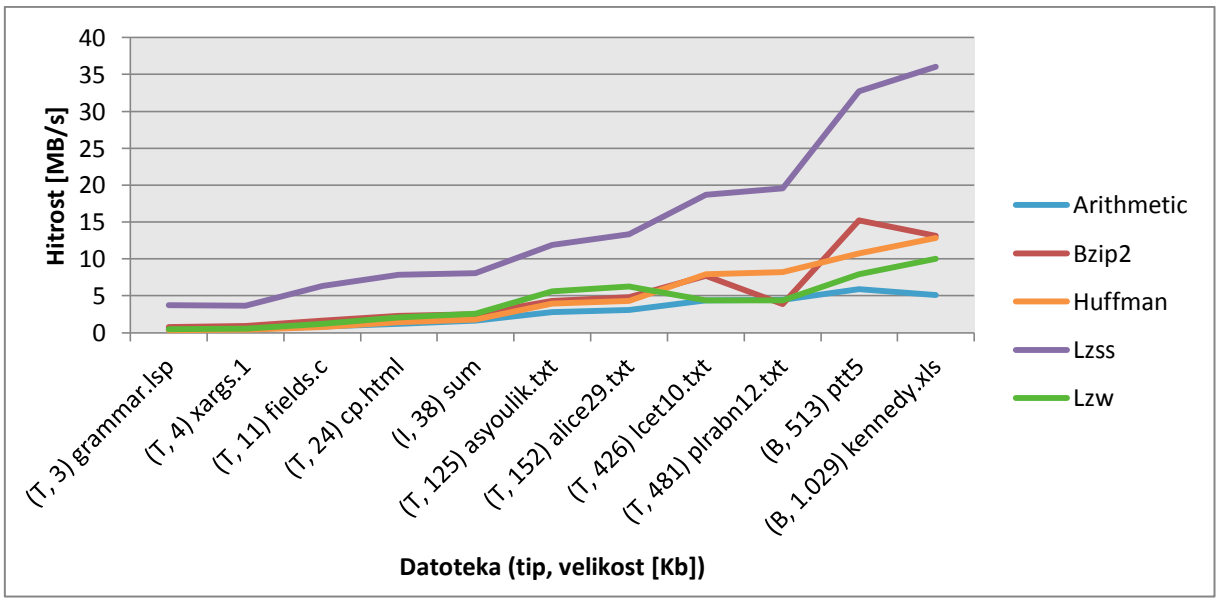



Slika 6.5: Grafični prikaz primerjave razmerij stiskanja za zbirko Canterbury.

Iz Tabela 6.10 razberemo, da je algoritem Lzss pri razširjanju podatkov ponovno najhitrejši. Na Slika 6.6 opazimo tudi, da se hitrost razširjanja pri Lzss postopoma povečuje skupaj z velikostjo datoteke. Edini večji padec hitrosti je očitno pri Bzip2 za datoteko plrabn12.txt.

| | Čas razširjanja [ms] | | | | |
|---|---|---|---|---|---|
| Datoteka | Arithmetic | Bzip2 | Huffman | Lzss | Lzw |
| grammar.lsp | 9,4 | 4,6 | 10,8 | **1,0** | 7,8 |
| xargs.1 | 9,4 | 4,6 | 11,2 | **1,1** | 8,0 |
| fields.c | 13,4 | 6,7 | 14,0 | **1,7** | 9,5 |
| cp.html | 20,4 | 10,6 | 17,7 | **3,1** | 12,0 |
| sum | 23,4 | 15,1 | 21,6 | **4,7** | 14,6 |
| asyoulik.txt | 45,2 | 29,2 | 31,6 | **10,5** | 22,4 |
| alice29.txt | 49,8 | 31,6 | 35,0 | **11,3** | 24,3 |
| lcet10.txt | 97,9 | 55,3 | 53,7 | **22,8** | 97,4 |
| plrabn12.txt | 107,8 | 124,2 | 58,6 | **24,5** | 110,7 |
| ptt5 | 87,3 | 33,6 | 47,8 | **15,6** | 65,0 |
| kennedy.xls | 202,6 | 78,4 | 80,2 | **28,5** | 102,7 |

Tabela 6.10: Rezultati časov razširjanja za zbirko Canterbury.



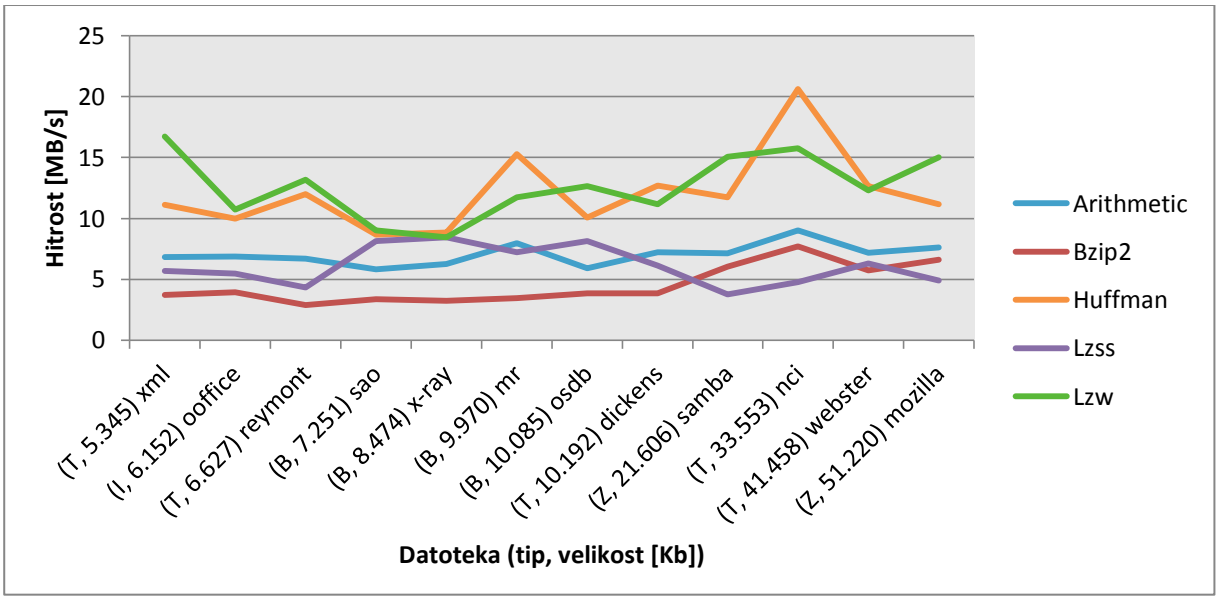

Slika 6.6: Grafični prikaz primerjave hitrosti razširjanja za zbirko Canterbury.

### 6.5.3 Analiza rezultatov na zbirki Silesia

V Tabela 6.11 vidimo, da se Huffmanovo kodiranje in algoritem Lzw izmenjujeta za imetnika najhitrejšega časa. V prejšnjih zbirkah je algoritmu Lzss uspelo imeti najhitrejši čas za manjše velikosti datotek, a ker so tokratne datoteke v zbirki Silesia precej večje, opazimo na Slika 6.7, da se ne obnese enako dobro za večje velikosti datotek kot Huffman in Lzw. Tudi tukaj je Bzip2 v večini primerov najpočasnejši, vendar skupaj z aritmetičnim kodiranjem ohranjata precej enakomerno hitrost pri vseh datotekah.

| | Čas stiskanja [ms] | | | | |
|---|---|---|---|---|---|
| Datoteka | Arithmetic | Bzip2 | Huffman | Lzss | Lzw |
| xml | 783,7 | 1.444,0 | 479,8 | 940,1 | **319,1** |
| ooffice | 898,0 | 1.569,3 | 616,7 | 1.122,1 | **574,0** |
| reymont | 989,0 | 2.284,1 | 553,2 | 1.522,8 | **503,0** |
| sao | 1.249,9 | 2.152,5 | 836,2 | 891,9 | **803,8** |
| x-ray | 1.358,5 | 2.628,0 | **959,7** | 1.002,5 | 1.001,4 |
| mr | 1.249,5 | 2.872,1 | **652,3** | 1.381,0 | 849,2 |
| osdb | 1.709,8 | 2.628,8 | 1.002,6 | 1.241,6 | **796,2** |
| dickens | 1.409,1 | 2.642,8 | **801,3** | 1.662,1 | 913,2 |
| samba | 3.033,5 | 3.565,4 | 1.839,0 | 5.730,7 | **1.433,2** |
| nci | 3.714,5 | 4.350,9 | **1.624,7** | 7.060,2 | 2.130,6 |
| webster | 5.768,4 | 7.231,9 | **3.273,2** | 6.599,1 | 3.366,6 |
| mozilla | 6.728,1 | 7.764,0 | 4.581,7 | 10.494,4 | **3.407,6** |

Tabela 6.11: Rezultati časov stiskanja za zbirko Silesia.

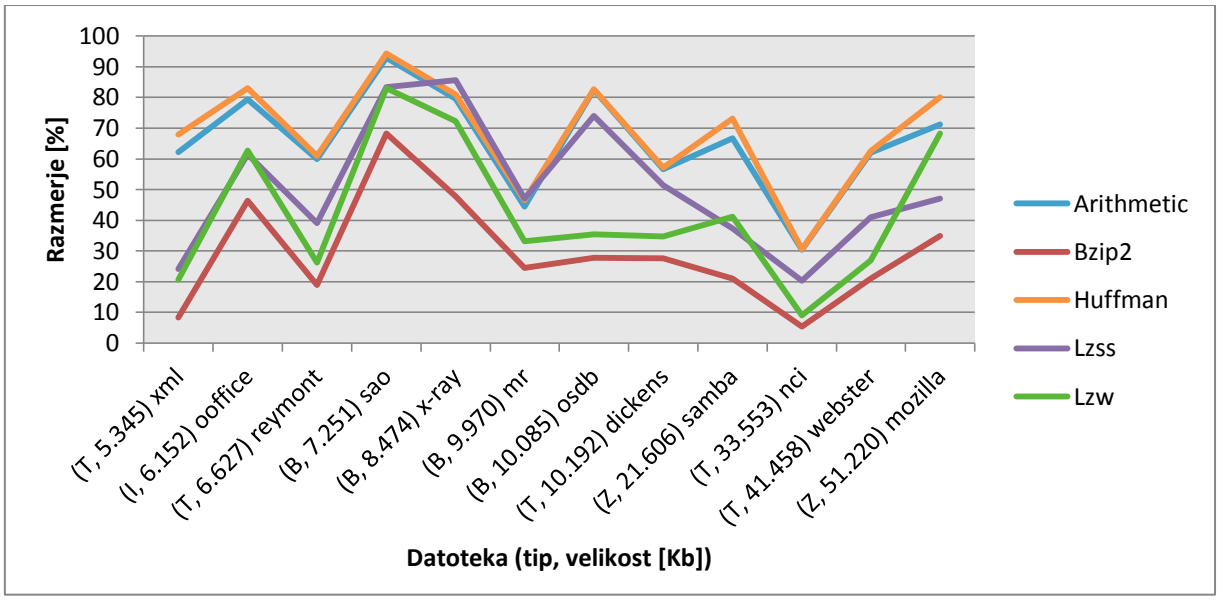



Slika 6.7: Grafični prikaz primerjave hitrosti stiskanja za zbirko Silesia.

Iz Tabela 6.12 razberemo, da ima Bzip2 najboljša razmerja pri vseh datotekah. Na Slika 6.8 vidimo, da je imelo Huffmanovo kodiranje skoraj povsod najslabše razmerje, sledilo mu je aritmetično kodiranje. Ker zbirka Silesia vsebuje več binarnih in združenih datotek, opazimo, da pri nekaterih datotekah (oofice, sao, osdb, samba, mozilla) algoritmi dosegajo manjše razmerje kot pri tekstovnih datotekah.

| | Razmerje [%] | | | | |
|---|---|---|---|---|---|
| Datoteka | Arithmetic | Bzip2 | Huffman | Lzss | Lzw |
| xml | 62,22 | **8,29** | 67,93 | 24,21 | 20,73 |
| ooffice | 79,31 | **46,44** | 82,96 | 61,49 | 62,78 |
| reymont | 59,93 | **18,95** | 60,91 | 39,1 | 26,23 |
| sao | 92,95 | **68,26** | 94,29 | 83,33 | 83,05 |
| x-ray | 79,57 | **47,8** | 80,88 | 85,57 | 72,3 |
| mr | 44,38 | **24,48** | 46,53 | 47,14 | 33,23 |
| osdb | 82,45 | **27,81** | 82,72 | 73,99 | 35,44 |
| dickens | 56,56 | **27,61** | 57,14 | 51,48 | 34,72 |
| samba | 66,62 | **21,09** | 73,14 | 37,36 | 41,23 |
| nci | 30,42 | **5,44** | 30,49 | 20,28 | 8,99 |
| webster | 61,94 | **21** | 62,46 | 41,03 | 26,92 |
| mozilla | 71,2 | **34,92** | 80 | 47,06 | 68,26 |

Tabela 6.12: Rezultati razmerij stiskanja za zbirko Silesia.

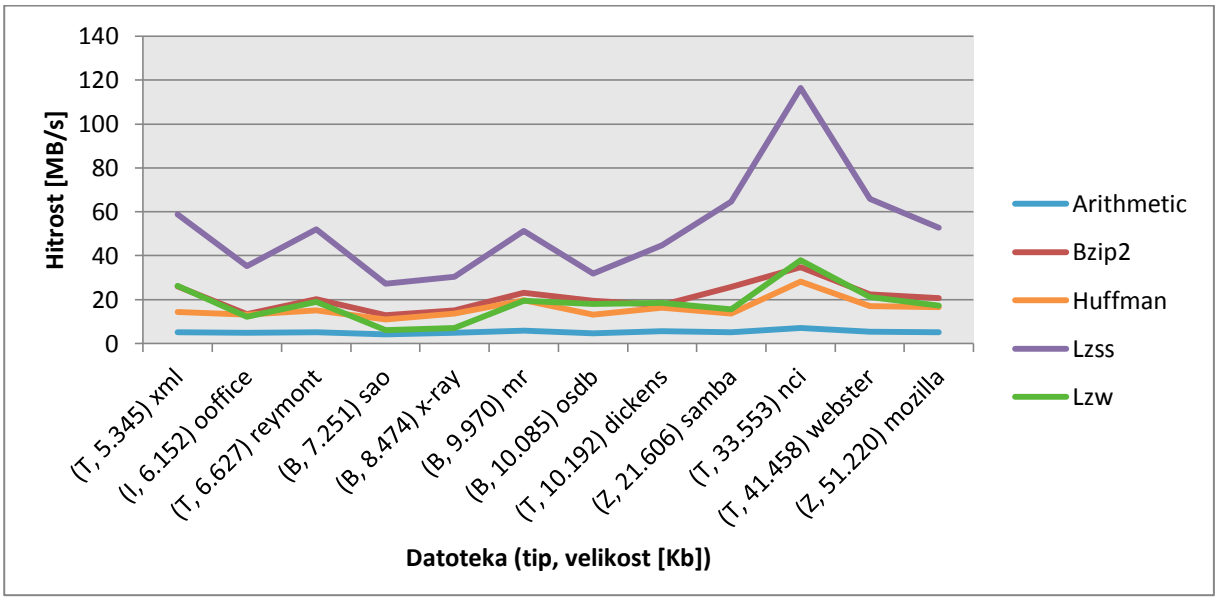



Slika 6.8: Grafični prikaz primerjave razmerij stiskanja za zbirko Silesia.

Tabela 6.13 prikazuje, da je algoritem Lzss še zmeraj najhitrejši pri razširjanju. Kljub precej večjim datotekam na Slika 6.9 opazimo, da Lzss dosega precej višje hitrosti razširjanja za razliko od prejšnjih dveh zbirk, zlasti pri datoteki nci. Čeprav je aritmetično kodiranje najpočasnejše, vseeno vzdržuje konstantno hitrost.

| | Čas razširjanja [ms] | | | | |
|---|---|---|---|---|---|
| Datoteka | Arithmetic | Bzip2 | Huffman | Lzss | Lzw |
| xml | 1.040,5 | 206,4 | 372,9 | **90,7** | 202,9 |
| ooffice | 1.297,5 | 458,8 | 465,7 | **174,9** | 502,0 |
| reymont | 1.284,8 | 329,3 | 438,9 | **127,2** | 350,4 |
| sao | 1.737,6 | 567,5 | 663,6 | **266,0** | 1.204,1 |
| x-ray | 1.784,0 | 560,3 | 626,9 | **278,6** | 1.195,4 |
| mr | 1.700,8 | 432,2 | 505,3 | **194,3** | 509,7 |
| osdb | 2.216,5 | 518,2 | 767,8 | **317,8** | 557,6 |
| dickens | 1.818,6 | 585,2 | 624,5 | **228,4** | 553,3 |
| samba | 4.168,8 | 838,9 | 1.595,6 | **334,6** | 1.395,7 |
| nci | 4.716,9 | 962,5 | 1.185,2 | **288,1** | 882,7 |
| webster | 7.727,7 | 1.859,2 | 2.455,0 | **628,4** | 1.964,0 |
| mozilla | 10.095,8 | 2.477,3 | 3.086,7 | **970,2** | 2.950,5 |

Tabela 6.13: Rezultati časov razširjanja za zbirko Silesia.



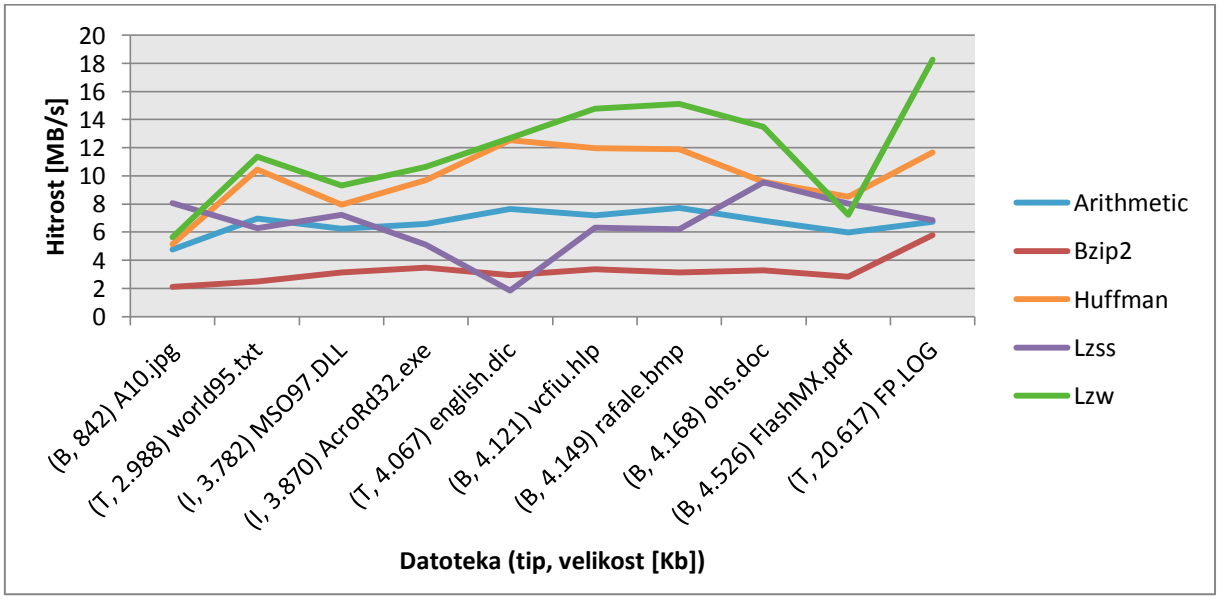

Slika 6.9: Grafični prikaz primerjave hitrosti razširjanja za zbirko Silesia.

### 6.5.4 Analiza rezultatov na zbirki Maximum

Iz Tabela 6.14 vidimo, da je algoritem Lzw najhitrejši, razen pri dveh datotekah. Na Slika 6.10 razberemo, da je zopet algoritem Bzip2 najpočasnejši, razen pri eni datoteki. Najbolj, kar odstopa iz grafa, sta Lzss pri datoteki english.dic in Lzw pri datoteki FlashMX.pdf, kjer se jima hitrost stiskanja znatno zmanjša. Podobno kot pri prejšnji zbirki aritmetično kodiranje in Bzip2 ves čas ohranjata bolj ali manj enako hitrost.

| | Čas stiskanja [ms] | | | | |
|---|---|---|---|---|---|
| Datoteka | Arithmetic | Bzip2 | Huffman | Lzss | Lzw |
| A10.jpg | 176,3 | 402,1 | 163,4 | **104,3** | 149,6 |
| world95.txt | 430,2 | 1.202,5 | 285,6 | 476,2 | **263,0** |
| MSO97.DLL | 607,0 | 1.202,8 | 476,0 | 522,6 | **406,2** |
| AcroRd32.exe | 588,4 | 1.114,1 | 400,0 | 759,7 | **363,9** |
| english.dic | 532,7 | 1.372,3 | 324,4 | 2.208,2 | **320,2** |
| vcfiu.hlp | 573,0 | 1.219,5 | 344,4 | 653,5 | **278,9** |
| rafale.bmp | 537,7 | 1.327,9 | 349,1 | 669,4 | **274,4** |
| ohs.doc | 611,8 | 1.265,4 | 434,6 | 436,0 | **308,6** |
| FlashMX.pdf | 758,0 | 1.597,3 | **531,1** | 563,9 | 624,6 |
| FP.LOG | 3.063,5 | 3.551,0 | 1.767,1 | 3.006,9 | **1.129,0** |

Tabela 6.14: Rezultati časov stiskanja za zbirko Maximum.

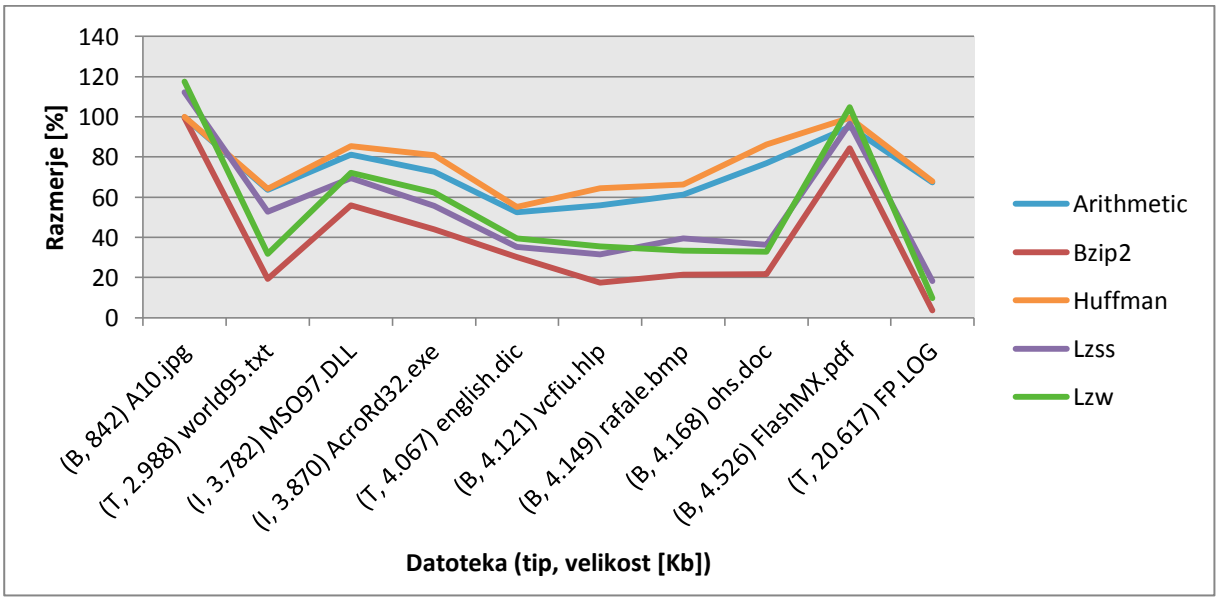



Slika 6.10: Grafični prikaz primerjave hitrosti stiskanja za zbirko Maximum.

Iz Tabela 6.15 vidimo, da je Bzip2 zopet najučinkovitejši pri stiskanju in ima najboljša razmerja. Na Slika 6.11 opazimo, da so vsi algoritmi dobili slabo razmerje pri datotekah A10.jpg in FlashMX.pdf, ponekod tudi čez 100 %, kar pomeni, da je prišlo do razširitve podatkov in s tem smo dejansko povečali velikost datoteke. Razlog za to je, da sta obe datoteki že stisnjeni z nekim algoritmom in večina naših algoritmov ne more še nadalje stisniti. Poleg tega vidimo, da so Lzss, Lzw in predvsem Bzip2 izredno dobro stisnili datoteko FP.LOG, do kar 3,52 odstotka originalne velikosti. Tako izredno razmerje algoritmi dosegajo zato, ker datoteka vsebuje dnevniške zapise in zato vsebuje zelo dolge in številčne ponovitve istih oziroma zelo podobnih nizov, ki jih je je možno izrabiti za stiskanje.

| | Razmerje [%] | | | | |
|---|---|---|---|---|---|
| Datoteka | Arithmetic | Bzip2 | Huffman | Lzss | Lzw |
| A10.jpg | 99,58 | **99,38** | 100,07 | 112,13 | 117,58 |
| world95.txt | 63,73 | **19,33** | 64,26 | 52,84 | 31,69 |
| MSO97.DLL | 81,19 | **55,88** | 85,48 | 69,32 | 72,21 |
| AcroRd32.exe | 72,61 | **43,9** | 80,8 | 55,72 | 62,15 |
| english.dic | 52,57 | **30,05** | 55,21 | 35,09 | 39,52 |
| vcfiu.hlp | 55,86 | **17,3** | 64,4 | 31,45 | 35,59 |
| rafale.bmp | 61,18 | **21,51** | 66,38 | 39,34 | 33,36 |
| ohs.doc | 76,76 | **21,75** | 86,22 | 36,27 | 32,81 |
| FlashMX.pdf | 95,05 | **84,28** | 99,38 | 96,4 | 104,81 |
| FP.LOG | 67,38 | **3,52** | 67,98 | 18,17 | 9,74 |

Tabela 6.15: Rezultati razmerij stiskanja za zbirko Maximum.

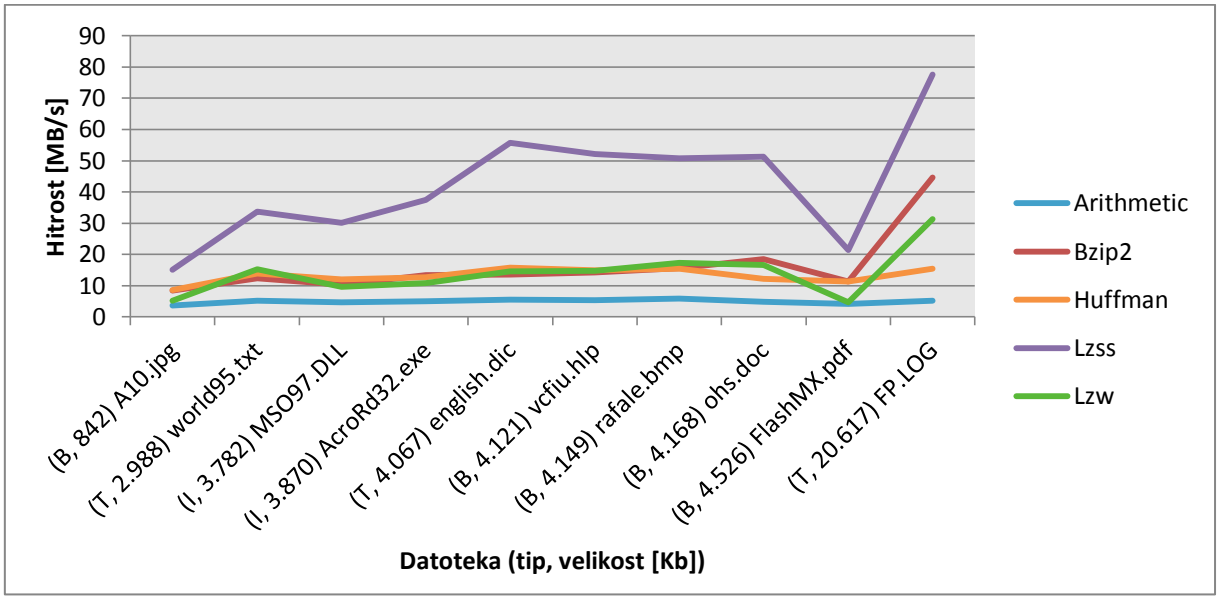



Slika 6.11: Grafični prikaz primerjave razmerij stiskanja za zbirko Maximum.

Iz Tabela 6.16 gotovo razberemo, da je Lzss znova najhitrejši pri razširjanju. Na Slika 6.12 opazimo, da je aritmetično kodiranje najpočasnejši med vsemi algoritmi, a kljub temu skupaj s Huffmanovim kodiranjem ohranjata enako hitrost, ne glede na velikosti ali tip datotek. Tudi tukaj opazimo vpliv že stisnjenih datotek A10.jpg in povsem FlashMX.pdf, kjer algoritmi Bzip2, Lzss in Lzw porabijo več časa.

| | Čas razširjanja [ms] | | | | |
|---|---|---|---|---|---|
| Datoteka | Arithmetic | Bzip2 | Huffman | Lzss | Lzw |
| A10.jpg | 226,3 | 100,4 | 97,7 | **55,9** | 165,1 |
| world95.txt | 576,4 | 241,2 | 216,2 | **88,6** | 195,8 |
| MSO97.DLL | 822,4 | 368,8 | 316,6 | **125,8** | 394,2 |
| AcroRd32.exe | 784,0 | 288,8 | 305,7 | **103,3** | 355,3 |
| english.dic | 745,1 | 298,8 | 256,8 | **73,0** | 278,3 |
| vcfiu.hlp | 758,4 | 288,9 | 277,6 | **79,0** | 278,8 |
| rafale.bmp | 699,9 | 266,8 | 270,3 | **81,6** | 240,1 |
| ohs.doc | 870,5 | 226,0 | 341,3 | **81,3** | 250,2 |
| FlashMX.pdf | 1.082,5 | 399,4 | 400,1 | **212,2** | 985,8 |
| FP.LOG | 4.009,3 | 462,7 | 1.335,4 | **265,8** | 658,6 |

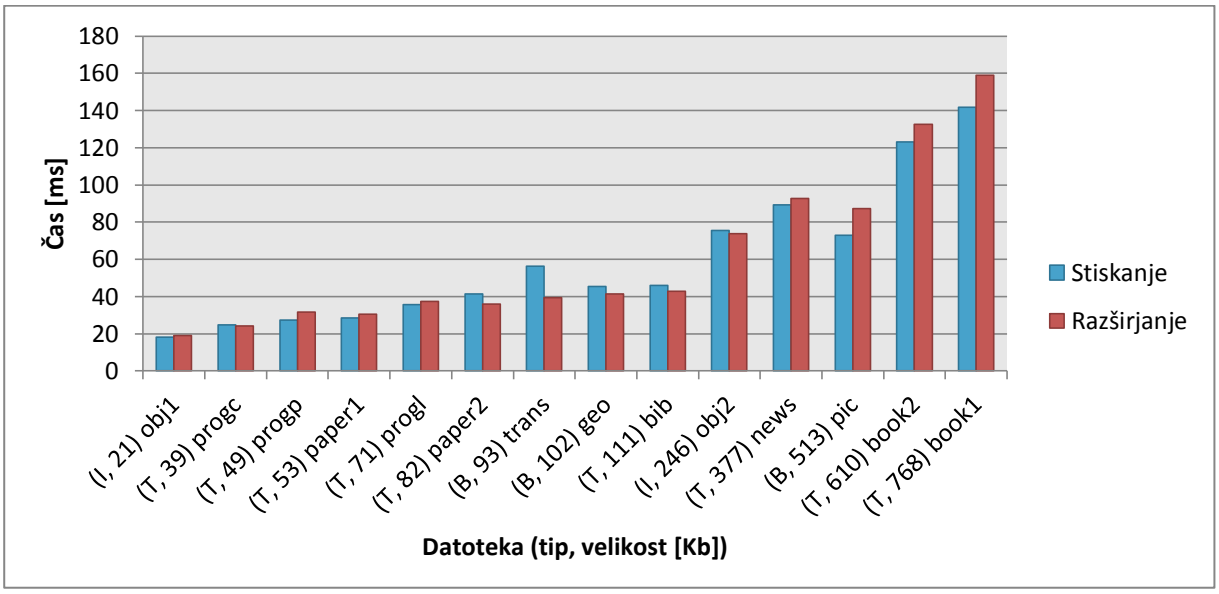
Tabela 6.16: Rezultati časov razširjanja za zbirko Maximum.



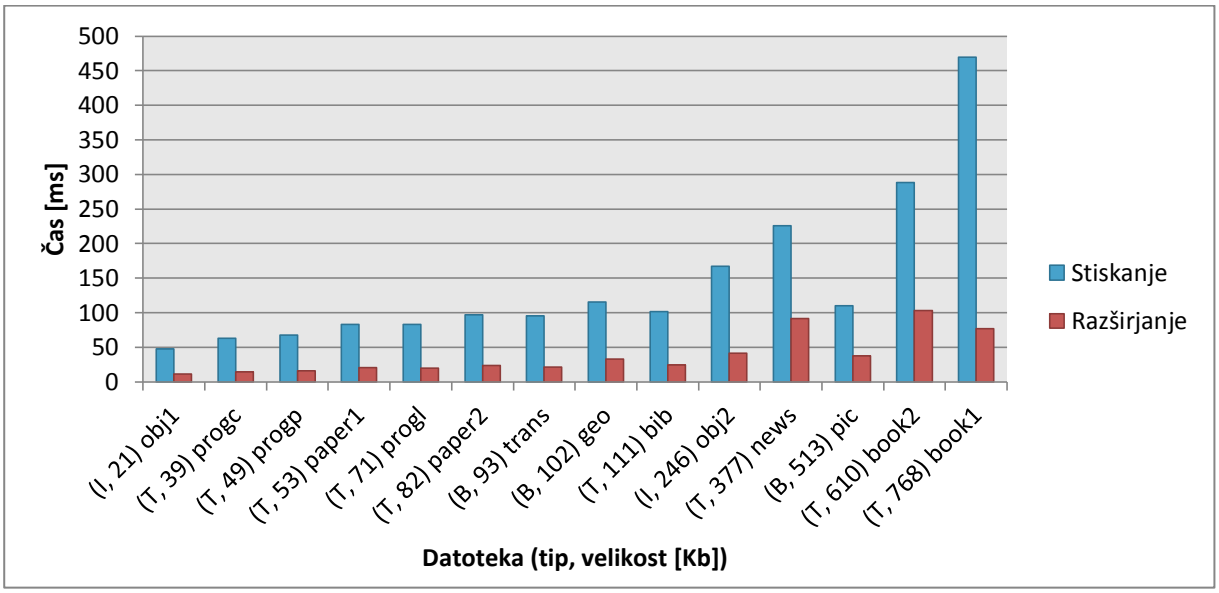

Slika 6.12: Grafični prikaz primerjave hitrosti razširjanja za zbirko Maximum.

### 6.5.5 Primerjava med časom stiskanja in razširjanja

Dosedanje analize so primerjale čase stiskanja in razširjanja posamezno, vendar je prav tako pomembno primerjati ta dva časa med seboj za posamezne algoritme. Slika 6.13 prikazuje čas stiskanja in čas razširjanja za aritmetično kodiranje na zbirki Calgary. Vidimo, da sta časa zelo podobna in skoraj enakovredna, kar je smiselno, saj je dekodiranje obratni postopek kodiranja. Na spodnji Slika 6.14 je prikazana primerjava za Bzip2. V tem grafu je razvidno, da je čas stiskanja večji od časa razširjanja, tudi do šestkrat večji pri zadnji datoteki book1.

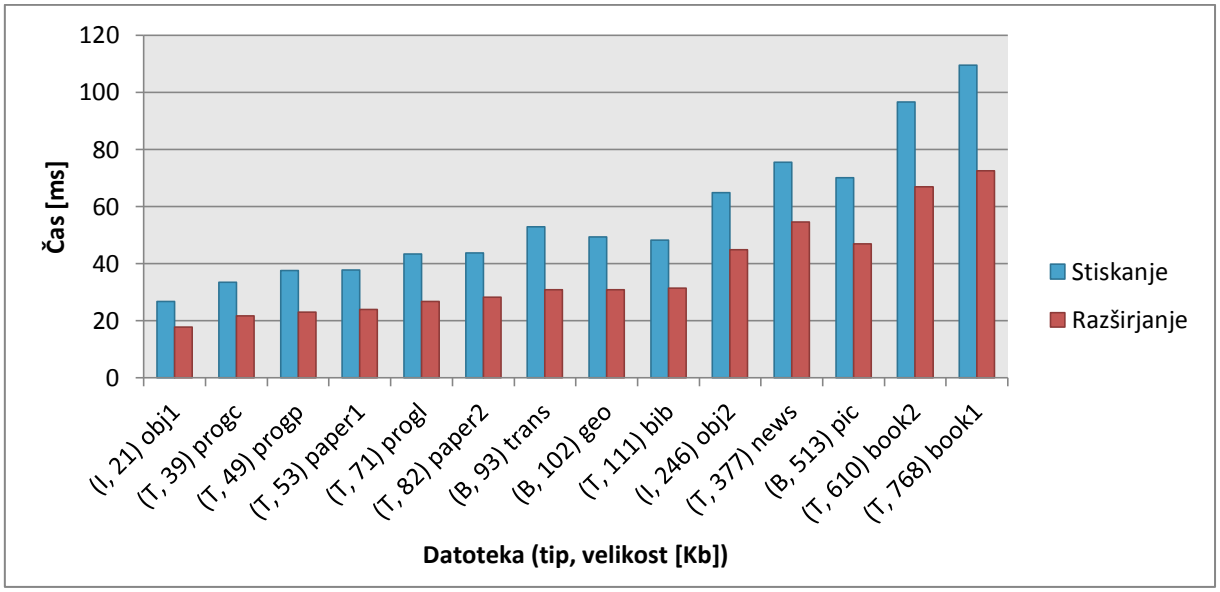



Slika 6.13: Grafični prikaz primerjave časov aritmetičnega kodiranja za zbirko Calgary.

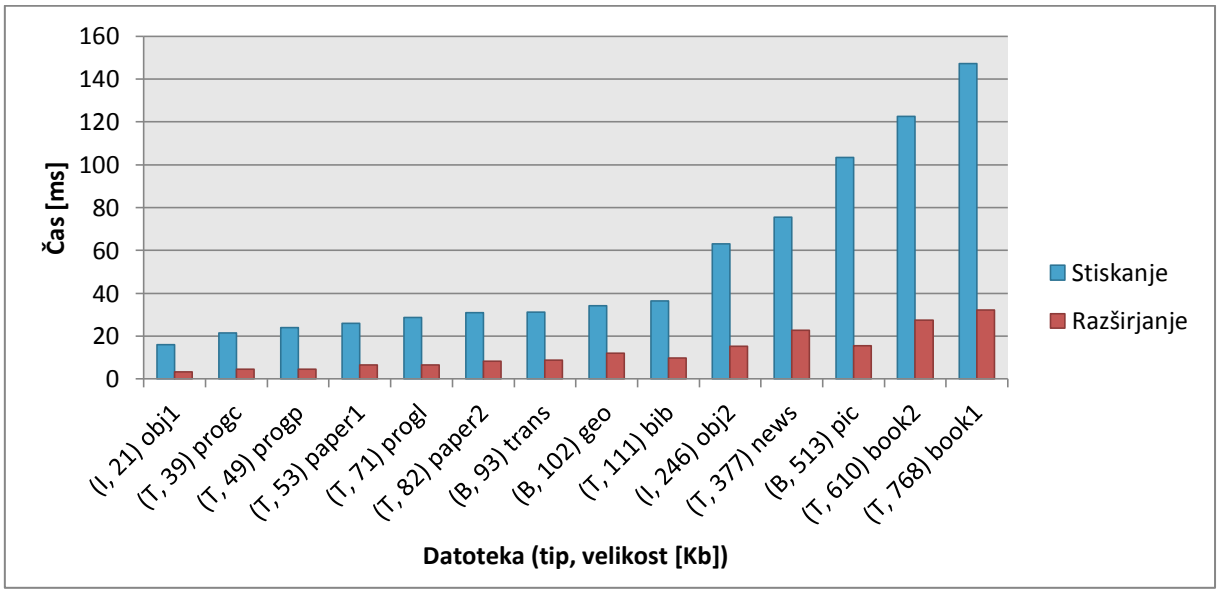



Slika 6.14: Grafični prikaz primerjave časov algoritma Bzip2 za zbirko Calgary.

Na Slika 6.15 vidimo primerjavo za Huffmanovo kodiranje, kjer vidimo, da je čas stiskanja večji od časa razširjanja, in sicer skoraj za polovico. Razlog za to se verjetno kaže v tem, da ima dekodirnik že zgrajeno drevo, medtem ko ga mora kodirnik sam izgraditi. Slika 6.16 prikazuje primerjavo za Lzss, kjer je očitno, da stiskanje vzame več časa kot razširjanje. Sicer je Lzss, skupaj z ostalimi različicami LZ77, znan po tem dejstvu, da ima čas razširjanja precej krajši od stiskanja.

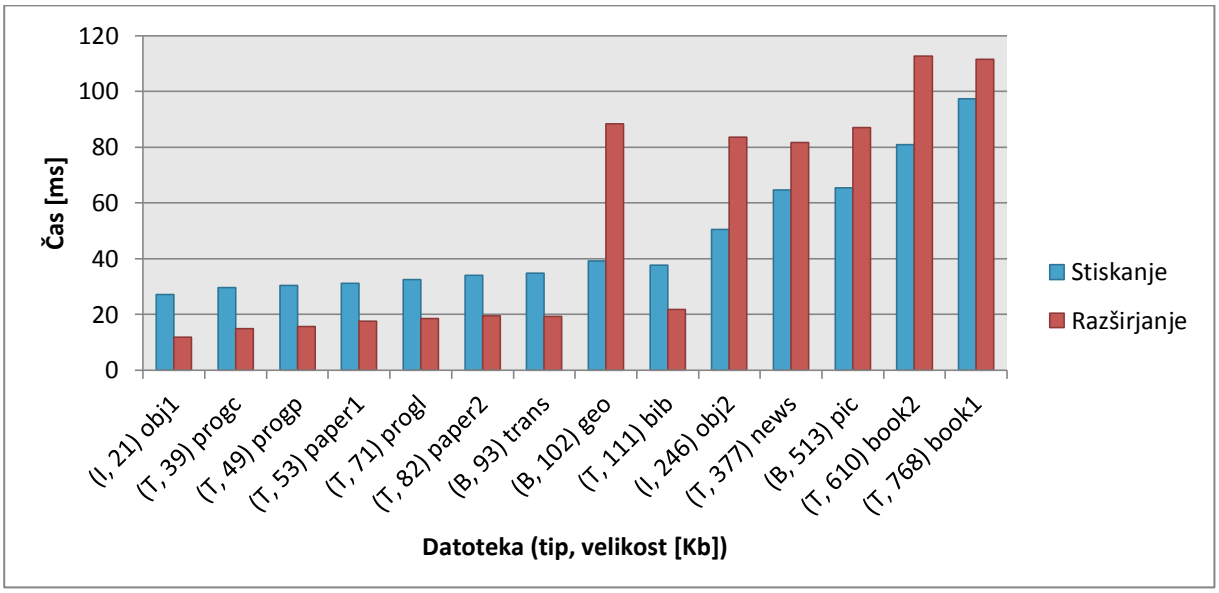



Slika 6.15: Grafični prikaz primerjave časov Huffmanovega kodiranja za zbirko Calgary.




Slika 6.16: Grafični prikaz primerjave časov algoritma Lzss za zbirko Calgary.

Slika 6.17 prikazuje primerjavo za Lzw, pri čemer opazimo zanimivo dejstvo, da časi niso enakomerno porazdeljeni, saj so na začetku časi stiskanja večji od razširjanja in proti koncu je ravno obratno.




Slika 6.17: Grafični prikaz primerjave časov algoritma Lzw za zbirko Calgary.

# 7 Sklepne ugotovitve

Diplomsko delo smo vsebinsko razdelili na dva dela. Za boljše razumevanje poznejših razlag algoritmov smo v teoretičnem delu začeli z lastnostmi kodiranja. Nato smo prešli na algoritme in opisali delovanja Shannon-Fanojevega kodiranja, Huffmanovega kodiranja in aritmetičnega kodiranja. Nadaljevali smo z algoritmi stiskanja s pomočjo slovarja, kjer smo najprej predstavili prvotna algoritma LZ77 in LZ78 in njuni različici LZSS in LZW. Zadnji algoritem, ki smo ga predstavili, je bil algoritem Burrows-Wheeler.

Nato smo se v teoretičnem delu osredotočili na testiranje algoritmov za stiskanje podatkov. Za testiranje smo uporabili sistem ALGator, v katerem smo uporabili odprtokodne implementacije za Huffmanovo kodiranje, aritmetično kodiranje, Lzss, Lzw in Bzip2. Izbrali smo algoritem Bzip2, ker je najpopularnejša implementacija algoritma Burrows-Wheeler. Implementacije algoritmov smo testirali na standardnih testnih zbirkah Calgary, Canterbury, Silesia in Maximum.

Najprej smo si pogledali rezultate časov stiskanja na zbirki Calgary. Tukaj nam je graf pokazal, da se hitrost stiskanja povečuje skupaj z velikostjo datotek pri vseh algoritmih. Isti primer naraščanja hitrosti smo imeli v naslednji testni zbirki Canterbury. V preostalih dveh zbirkah pa so datoteke precej večje, kjer hitrosti niso več naraščale. Tu so imeli Huffmanovo kodiranje, Lzss in Lzw različne hitrosti, medtem ko sta aritmetično kodiranje in Bzip2 ohranjala precej enakomerno hitrost. Zanimivi sta datoteki pic in ptt5, ki sta pravzaprav identični, kjer smo videli največjo porast hitrosti za algoritem Bzip2. Datoteki večinoma sestojita iz bajtov dolžine 0 in ker algoritmu Bzip2 koristi imeti veliko število ničtih bajtov, zaradi faze RLE stiskanja, je logično, da se pri teh dveh datotekah zelo dobro izkaže. Pri manjših datotekah je bil najhitrejši Lzss, pri večjih datotekah pa sta bila najhitrejša Huffmanovo kodiranje in Lzw.

Iz vseh štirih testnih zbirk se je takoj izkazalo, da je Lzss nedvomno najhitrejši pri razširjanju vseh datotek. V večini primerov je več kot trikrat hitrejši od ostalih. To dejstvo ne preseneča, saj nima kompleksnega postopka dekodiranja v primerjavi z ostalimi algoritmi, poleg tega pa za svojo podatkovno strukturo uporablja le medpomnilnik v obliki krožne vrste. Na nasprotnem koncu smo opazili, da je bilo aritmetično kodiranje najpočasnejše, predvsem pri zbirkah Silesia in Maximum. Kljub temu da je aritmetično kodiranje bolj ali manj

najpočasnejše, ima najbolj enakomerno hitrost razširjanja pri vseh datotekah, ne glede na velikost.

Pri preverjanju rezultatov razmerij stiskanja je bilo očitno, da ima Bzip2 najboljša razmerja pri vseh datotekah zbirk, medtem ko ima Huffmanovo kodiranje najslabša razmerja. Huffmanovemu kodiranju se najbolj približa aritmetično kodiranje, kar tudi potrjuje teorijo, da proizvaja aritmetično kodiranje bolj optimalne kode. Sicer pa so razmerja zelo sorazmerna med vsemi algoritmi. Edina izjema je datoteka geo, pri kateri se razmerja poslabšajo za algoritme Bzip2, Lzss in Lzw, toda za aritmetično kodiranje in Huffmanovo kodiranje se razmerje ne spreminja. Datoteke takega formata so torej primerne za statistične algoritme. Pri zbirki Maximum smo ugotovili, da sta datoteki A10.jpg in FlashMX.pdf že stisnjeni, kar tudi odraža na razmerju teh dveh datotek, kajti razmerja so blizu 100 odstotkov. Pri datoteki A10.jpg je Lzss povečal datoteko kar za 12 odstotkov in Lzw za 17 odstotkov, toda za datoteko FlashMX.pdf je samo Lzw povečal, in sicer za 4 odstotke. Tak primer razširitve podatkov ni zaželen pri algoritmih za stiskanje podatkov, zato je dobro, če taki algoritmi zagotovijo lastnost, da nikoli ne povečajo dejanske velikosti stisnjene datoteke.

Za konec smo še primerjali čase stiskanja in razširjanja med posameznimi algoritmi. Tukaj smo ugotovili, da ima aritmetično kodiranje oba časa podobna. Za Lzw opazimo, da ima neenakomerno razmerje med časom stiskanja in razširjanem. Za ostale tri algoritme pa izvemo, da imajo čas stiskanja precej večji od časa razširjanja, zlasti Bzip2 in Lzss.

Iz primerjav vidimo, da ima vsak algoritem svoje značilnosti in s tem je primernejši za drugačne situacije. Aritmetično kodiranje je primerno takrat, kadar potrebujemo algoritem, ki stiska in razširja s konstantno hitrostjo, ne glede na velikost datoteke, vendar je najpočasnejši pri razširjanju. Bzip2 je najboljši pri razmerju stiskanja, vendar ima najpočasnejšo hitrost stiskanja. Huffmanovo kodiranje se dobro pokaže pri hitrosti stiskanja večjih velikosti datotek, toda dosega najslabša razmerja stiskanja. Lzss je hiter pri stiskanju manjših velikosti datotek in še zlasti je najhitrejši pri razširjanju, zato je primeren tam, kjer se podatki neprestano razširjajo. Lzw je hiter pri stiskanju večjih datotek, vendar ima časa stiskanja in razširjanja zelo neenakomerna.

# Literatura


[1] R. Arnold in T. Bell, "A corpus for the evaluation of lossless compression algorithms", Data Compression Conference, 1997. DCC'97, 1997.

[2] T. Bell, I. H. Witten in J. G. Cleary, "Modeling for text compression", ACM Computing Surveys (CSUR), vol. 21, št. 4, str. 557–591, 1989.

[3] T. Bell, I. H. Witten in J. G. Cleary, "Text Compression", Prentice–Hall, Inc., 1990.

[4] J. L. Bentley, D. D. Sleator, R. E. Tarjan, V. K. Wei, "A Locally Adaptive Data Compression Scheme", Communications of the ACM, vol. 29, št. 4, 1986.

[5] L. Boltzmann, "Weitere Studien über das Wärmegleichgewicht unter Gasmolekülen." Sitzungsberichte Akademie der Wissenschaften, str. 275–370, 1872.

[6] M. Burrows in D. Wheeler, "A block–sorting lossless data compression algorithm", Digital Systems Research Center, št. 124, 1994.

[7] S. Deorowicz, "Universal lossless data compression algorithms", Philosophy Dissertation Thesis, 2003.

[8] R. M. Fano, "The Transmission of Information", Massachusetts Institute of Technology, Research Laboratory of Electronics, št. 65, 1949.

[9] R. G. Gallager, "Variations on a theme by Huffman", IEEE Transactions on Information Theory, vol. 24, št. 6, str. 668–674, 1978.

[10] D. A. Huffman, "A method for the construction of minimum–redundancy codes", Proceedings of the IRE, vol. 40, št. 9, str. 1098–1101, 1952.

[11] P. Jakopin, "Zgornja meja entropije pri leposlovnih besedilih v slovenskem jeziku", doktorska disertacija, 1999.

[12] M. Nelson in J. L. Gailly, "The data compression book", New York: M&t Books, vol. 2, 1995.

[13] I. M. Pu, "Fundamental data compression", Butterworth–Heinemann, 2005.

[14] D. Salomon in G. Motta, "Handbook of Data Compression", Springer, vol. 5, 2010.

[15] A. A. Sardinas in G. W. Patterson, "A necessary and sufficient condition for the unique decomposition of coded messages", Convention Record of the I.R.E., 1953 National Convention, Part 8: Information Theory, str. 104–108, 1953.

[16] C. E. Shannon, "A Mathematical Theory of Communication", Bell System Technical Journal, vol. 27, str. 379–423, 623–656, 1948.

[17] J. A. Storer in T. G. Szymanski, "Data compression via textual substitution", Journal of the ACM (JACM), vol. 29, št.4, str. 928–951, 1982.

[18] T. A. Welch, "A Technique for High–Performance Data Compression", IEEE Computer, vol. 17, št. 6, str. 8–19, 1984.

[19] J. G. Wolff, "Computing as compression: An overview of the SP theory and system", New Generation Computing, vol. 13, št. 2, str.187–214, 1995.

[20] J. Ziv in A. Lempel, "A universal algorithm for sequential data compression", IEEE Transactions on information theory, vol. 23, št.3, str. 337–343, 1977.

[21] J. Ziv in A. Lempel, "Compression of individual sequences via variable–rate coding", Information Theory, IEEE Transactions on, vol. 24, št. 5, str. 530–536, 1978.

[22] W. L. Eastman, A. Lempel, J. Ziv in M. Cohn, "Apparatus and method for compressing data signals and restoring the compressed data signals", št. patenta US4464650 (A), 7.08.1984.

[23] G.G. Langdon in J.J. Rissanen, "High–speed arithmetic compression coding using concurrent value updating", št. patenta US4467317 (A), 21.08.1984.

[24] G.G. Langdon in J.J. Rissanen, "Method and Means for Arithmetic Coding Utilizing a Reduced Number of Operations", št. patenta US4286256 (A), 25.08.1981.

[25] G.G. Langdon in J.J. Rissanen, "Method and Means for Arithmetic String Coding", št. patenta US4122440 (A), 24.10.1978.

[26] V. S. Miller in M. N. Wegman, "Data compression method", št. patenta US4814746 (A), 21.3.1989.

[27] T. A. Welch, "High speed data compression and decompression apparatus and method", št. patenta US4558302 (A), 10.12. 1985.

[28] Projekt Compression za sistem ALGator [online]. Dosegljivo: https://github.com/milos3/ALGator-Compression. [Dostopano: 12.3.2016].

[29] ALGator project [online]. Dosegljivo: https://github.com/ALGatorDevel/Algator. [Dostopano: 6.2.2016].

[30] History of Lossless Data Compression Algorithms [online]. Dosegljivo: http://ethw.org/History_of_Lossless_Data_Compression_Algorithms. [Dostopano: 10.1.2016].

[31] Lossless data compression software benchmarks / comparisons [online]. Dosegljivo: http://www.maximumcompression.com/. [Dostopano: 13.10.2015].

[32] LZSS (LZ77) Discussion and Implementation [online]. Dosegljivo: http://michael.dipperstein.com/lzss/. [Dostopano: 20.1.2016].

[33] Silesia compression corpus [online]. Dosegljivo: http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia. [Dostopano: 10.10.2015]

[34] Without Shannon's information theory there would have been no internet [online]. Dosegljivo: https://www.theguardian.com/science/2014/jun/22/shannon–information–theory. [Dostopano: 20.12.2015].